



WestminsterResearch

<http://www.westminster.ac.uk/westminsterresearch>

A generic framework for process execution and secure multi-party transaction authorization.

Thomas Weigold

School of Electronics and Computer Science

This is an electronic version of a PhD thesis awarded by the University of Westminster. © The Author, 2010.

This is an exact reproduction of the paper copy held by the University of Westminster library.

The WestminsterResearch online digital archive at the University of Westminster aims to make the research output of the University available to a wider audience. Copyright and Moral Rights remain with the authors and/or copyright owners.

Users are permitted to download and/or print one copy for non-commercial private study or research. Further distribution and any use of material from within this archive for profit-making enterprises or for commercial gain is strictly forbidden.

Whilst further distribution of specific materials from within this archive is forbidden, you may freely distribute the URL of WestminsterResearch:
(<http://westminsterresearch.wmin.ac.uk/>).

In case of abuse or copyright appearing without permission e-mail
repository@westminster.ac.uk

**A GENERIC FRAMEWORK FOR PROCESS
EXECUTION AND SECURE MULTI-PARTY
TRANSACTION AUTHORIZATION**

THOMAS WEIGOLD

A thesis submitted in partial fulfilment of
the requirements of the University of Westminster
for the degree of Doctor of Philosophy

This research programme was carried out
in collaboration with the
IBM Zurich Research Laboratory

March 2010

Table of Contents

List of Figures	5
List of Source Code Listings	7
List of Tables.....	8
List of Abbreviations	9
Acknowledgments	11
Declaration.....	12
Abstract.....	14
1. Introduction.....	15
1.1. Workflow and Business Process Management	15
1.2. Process-Driven Applications.....	18
1.3. Secure Remote Authentication and Transaction Authorization.....	20
1.4. Aim and Scope of this Thesis.....	23
1.5. Motivation.....	24
1.6. Methodology	25
1.7. Application Domain.....	26
1.8. Challenges and Requirements	28
1.8.1. Process Execution Framework Challenges	28
1.8.2. Secure Remote Multi-Party Transaction Authorization Challenges	33
1.9. Direction of Research and Thesis Organization.....	36
2. Related Work	38
2.1. Process Execution Frameworks.....	38
2.1.1. Introduction	38
2.1.2. IBM WebSphere Process Server	38
2.1.3. Windows Workflow Foundation	40
2.1.4. Java Business Process Management.....	47
2.1.5. Yet Another Workflow Language	50

Table of Contents

2.1.6.	Bossa.....	51
2.1.7.	OpenSymphony Workflow	53
2.2.	Authentication and Transaction Authorization Methods.....	55
2.2.1.	Introduction	55
2.2.2.	Remote Authentication Schemes.....	55
2.2.3.	Client Security Devices	60
2.2.4.	Threat-Solution Taxonomy	63
2.2.5.	Mobile Transaction Authentication Number	66
2.2.6.	Standalone Challenge/Response Reader.....	68
2.2.7.	Flicker Devices.....	69
2.2.8.	Secoder and Financial Transaction IC Card Reader.....	71
2.2.9.	Mobile Identity and Sentinel.....	72
2.3.	Summary and Discussion.....	73
3.	A Generic Framework for Process Execution.....	76
3.1.	High Level Architecture	76
3.2.	Process Model	79
3.3.	From CEFSMs to Process Definitions.....	83
3.3.1.	Process Definition Language	83
3.3.2.	Process Anatomy	84
3.4.	Concurrency and Synchronization.....	89
3.4.1.	ePVM Threads.....	89
3.4.2.	Process Types	92
3.5.	Communication	94
3.5.1.	Inter-Process Communication	94
3.5.2.	ePVM-Host Communication.....	97
3.6.	Monitoring.....	100
3.7.	Persistence.....	102

Table of Contents

3.7.1.	Introduction	102
3.7.2.	A flexible Persistence Model	103
3.7.3.	Persistence in ePVM.....	107
3.8.	The ePVM Prototype Implementation.....	110
3.8.1.	Overview	110
3.8.2.	Host Application Programming Interface.....	111
3.8.3.	Runtime Application Programming Interface	113
3.8.4.	The Timer Process	115
3.8.5.	The Stock Quote Example Application	115
3.9.	Summary and Contributions	119
4.	Secure Multi-Party Transaction Authorization.....	120
4.1.	The Zone Trusted Information Channel	120
4.1.1.	The Basic Approach	120
4.1.2.	In-Stream Usage	123
4.1.3.	Second Channel Usage	126
4.1.4.	Multi-Party Transaction Authorization.....	128
4.1.5.	Ease-of-Use and Mobility	129
4.1.6.	Administration and Integration	130
4.2.	Attacks and Remedies.....	131
4.3.	Prototypes.....	133
4.4.	Integration with ePVM	134
4.5.	Summary and Contributions	137
5.	Case Study and Evaluation.....	139
5.1.	ePVM Framework	139
5.1.1.	Basic Framework Validation.....	139
5.1.2.	Persistence.....	144
5.2.	The ZTIC.....	147

Table of Contents

5.3. The Biometric Identification Use Case.....	149
5.3.1. Introduction	149
5.3.2. The Grid Component Model Framework	150
5.3.3. Process-Driven Distributed Biometric Identification	154
5.3.4. Results, Experiences, and Lessons Learned.....	160
5.4. Summary and Contributions	164
6. Conclusions and Future Directions	165
6.1. Summary and Achievements.....	165
6.2. Conclusions	167
6.3. Future Directions	167
References	170
Appendix A. ePVM Host and Runtime API	180
A.1. ePVM Host API.....	180
A.2. ePVM Runtime API.....	181
Appendix B. BIS Initialization Process Definition	183
Appendix C. BIS Application GUI.....	186

LIST OF FIGURES

Figure 1.	Workflow reference model – components & interfaces.....	16
Figure 2.	The BPM lifecycle	18
Figure 3.	Process-driven applications vs. BPMS/WfMS	19
Figure 4.	The three main types of attack scenarios.....	21
Figure 5.	Transaction authentication challenges	33
Figure 6.	WebSphere BPM products	39
Figure 7.	WebSphere process server platform overview	39
Figure 8.	Workflow foundation components.....	42
Figure 9.	A sequential workflow example drawn in Visual Studio.....	43
Figure 10.	The jBPM framework architecture.....	47
Figure 11.	Symbols used in YAWL	50
Figure 12.	The four primary authentication schemes.....	58
Figure 13.	Authentication thread-solution taxonomy.....	65
Figure 14.	mTAN usage scenario.....	67
Figure 15.	Barclays EMV CAP reader	68
Figure 16.	Flickering communication with AXS-Card	70
Figure 17.	FINREAD compliant class 4 smart card reader	71
Figure 18.	Kobil mIDentity USB stick.....	73
Figure 19.	ePVM top level architectural design	77
Figure 20.	UML state diagram of a computer keyboard state machine [101]	83
Figure 21.	Event-driven invocation of the process function.....	85
Figure 22.	An ePVM process instance of type <i>HANDLER</i>	93
Figure 23.	Inter-process communication	95
Figure 24.	ePVM-host communication	99
Figure 25.	Conceptual overview and example flow.....	105
Figure 26.	Asynchronous persistence operations via PDH	109
Figure 27.	The ePVM prototype implementation	111
Figure 28.	Stock quote application architecture.....	116
Figure 29.	The stock quote process flow	117
Figure 30.	Basic ZTIC configuration	121
Figure 31.	ZTIC in-stream usage scenario	124
Figure 32.	ZTIC second channel usage scenario	128

List of Figures

Figure 33.	ZTIC device early prototype	133
Figure 34.	ZTIC device production prototype	134
Figure 35.	ePVM multi-party transaction authorization scenario.....	136
Figure 36.	The ZTAN ePVM extension	137
Figure 37.	CPU usage executing 200 stock quote instances.....	142
Figure 38.	Memory usage executing 200 stock quote instances.....	143
Figure 39.	Java heap memory usage during experiment	146
Figure 40.	ZTIC internet bank application	148
Figure 41.	Behavioural skeleton rationale	152
Figure 42.	Behavioural skeletons implemented in GCM	153
Figure 43.	BIS high-level architecture	155
Figure 44.	BIS initialization process flow	156
Figure 45.	BIS using the data-parallel BS	158
Figure 46.	BIS source code breakdown.....	162

LIST OF SOURCE CODE LISTINGS

Listing 1.	Windows workflow foundation activity rule snippet	31
Listing 2.	Code-only WF workflow code snippet	44
Listing 3.	Code-separation XAML workflow markup	45
Listing 4.	jBPM loan approval example using ProcessFactory	49
Listing 5.	Creating a Bossa case type.....	53
Listing 6.	OSWorkflow XML workflow markup snippet	54
Listing 7.	A simple ePVM process definition.....	86
Listing 8.	ePVM internal process and package anatomy	88
Listing 9.	Event-driven threading example	90
Listing 10.	Order management example process	96
Listing 11.	Host application/process example	100
Listing 12.	Persistence service messages (JSON with JavaDoc).....	107
Listing 13.	Flexible persistence example (snippet).....	110
Listing 14.	Stock quote process definition	119
Listing 15.	ePVM process simulating real-world business process.....	145

LIST OF TABLES

Table 1.	ePVM host API methods.....	112
Table 2.	ePVM runtime API functions	113
Table 3.	Summary: Challenges and ZTIC solutions.....	138
Table 4.	Stock quote example source code break down	140
Table 5.	Execution performance of basic operations.....	144

LIST OF ABBREVIATIONS

ABC	Autonomic Behaviour Controller
ADL	Architecture Description Language
AM	Autonomic Manager
API	Application Programming Interface
BIS	Biometric Identification System
BPEL	Business Process Execution Language
BPM	Business Process Management
BPMN	Business Process Management Notation
BPMS	Business Process Management System
BS	Behavioural Skeleton
CA	Certification Authority
CAP	Card Authentication Program
CEFSM	Communicating Extended Finite State Machines
EFSM	Extended Finite State Machines
EMV	Europay, MasterCard, Visa
ePVM	Embeddable Process Virtual Machine
GCM	Grid Component Model
J2EE	Java 2 Enterprise Edition
jBPM	JBoss Java Business Process Management
JSON	JavaScript Object Notation
KB	Kilobyte
MB	Megabyte
MDD	Model Driven Development
MITM	Man-in-the-middle
MSD	Mass Storage Device
MSW	Malicious software
mTAN	Mobile Transaction Authentication Number
PDH	Persistence Device Handler
PIN	Personal Identification Number
PKI	Public Key Infrastructure
QoS	Quality of Service

List of Abbreviations

SLA	Service Level Agreement
SMS	Short Message Service
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Socket Layer
TAN	Transaction Authentication Number
TLS	Transport Layer Security
USB	Universal Serial Bus
WF	(Windows) Workflow Foundation
WfMC	Workflow Management Coalition
WfMS	Workflow Management System
WPS	WebSphere Process Server
WS	Web Service
WSDL	Web Service Description Language
XAML	Extensible Application Markup Language
XML	Extensible Markup Language
XPDL	XML Process Definition Language
YAWL	Yet Another Workflow Language
ZTAN	ZTIC Transaction Authentication Number
ZTIC	Zone Trusted Information Channel

ACKNOWLEDGMENTS

First and foremost, I would like to thank my Director of Studies Professor Vladimir Getov and the University of Westminster. Special thanks also go to my supervisor Dr. Peter Buhler and the IBM Zurich Research Laboratory.

Finally, and most importantly, I am forever indebted to the most important people in my life, Rosemarie, Paul, Stefanie, and Lenny, for their continuous support. I dedicate this Thesis to them.

DECLARATION

Some parts of the work presented in this thesis have been published in the following journals, conferences, and patents:

Journals

- T. Weigold, M. Aldinucci, M. Danelutto, V. Getov, "Process-Driven Biometric Identification by means of Autonomic Grid Components", Int. Journal of Autonomous and Adaptive Communications Systems (IJAACS), Inderscience, 2010 (to appear).
- T. Weigold, T. Kramp, M. Baentsch, "Remote Client Authentication", IEEE Security & Privacy Journal, July-August 2008, Volume 6, Issue 4, pp. 36-43.
- A. Hiltgen, T. Kramp, T. Weigold, "Secure Internet Banking Authentication", IEEE Security and Privacy Journal, vol. 4, no. 2, Mar/Apr, 2006, pp. 21-29.

Conference Proceedings

- T. Weigold, T. Kramp, P. Buhler, "Flexible Persistence Support for State Machine-based Workflow Engines", Proc. 4th IEEE International Conference on Software Engineering Advances (ICSEA), 20-25 September, 2009, pp. 313-319.
- T. Weigold, M. Aldinucci, M. Danelutto, V. Getov. "Integrating Autonomic Grid Components and Process-Driven Business Applications", Proc. 3rd International ICST Conference on Autonomic Computing and Communication Systems (Autonomics), 9-11 September, 2009, pp. 98-115.
- A. Basukoski, P. Buhler, V. Getov, S. Isaiadis, T. Weigold, "Methodology for Component-based Development of Grid Applications", Proc. Workshop on Component Based High Performance Computing (CBHPC), ACM Digital Library, October 14-17, 2008.
- T. Weigold, P. Buhler, A. Basukoski, V. Getov, "Advanced Grid Programming with Components: A Biometric Identification Case Study", Proc. 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC), July 28 - August 1, 2008, pp. 401-408.
- T. Weigold, T. Kramp, R. Hermann, F. Hoering, P. Buhler, M. Baentsch. "The Zurich Trusted Information Channel - An Efficient Defense Against Man-in-the-middle and

Malicious Software Attacks", in P. Lipp, A.-R. Sadeghi, and K.-M. Koch (Eds.): Proc. TRUST 2008, LNCS 4968, pp. 75–91, 2008, Springer-Verlag, 2008.

- T. Weigold, T. Kramp, P. Buhler, “ePVM – An Embeddable Process Virtual Machine”, Proc. 31st Annual IEEE International Computer Software and Applications Conference (COMPSAC), July 23-27, 2007, pp. 557-564.

Patents

- WO 2009/122360 A2, 8th October 2009, “Secure Online Banking Transactions”
- WO 2009/066217 A2, 21st May 2009, “Performing Secure Electronic Transactions”
- WO 2010/032207 A1, 25th March 2010, “Authorization of Server Operations”

ABSTRACT

Process execution engines are not only an integral part of workflow and business process management systems but are increasingly used to build process-driven applications. In other words, they are potentially used in all kinds of software across all application domains. However, contemporary process engines and workflow systems are unsuitable for use in such diverse application scenarios for several reasons. The main shortcomings can be observed in the areas of interoperability, versatility, and programmability. Therefore, this thesis makes a step away from domain specific, monolithic workflow engines towards generic and versatile process runtime frameworks, which enable integration of process technology into all kinds of software. To achieve this, the idea and corresponding architecture of a generic and embeddable process virtual machine (ePVM), which supports defining process flows along the theoretical foundation of communicating extended finite state machines, are presented. The architecture focuses on the core process functionality such as control flow and state management, monitoring, persistence, and communication, while using JavaScript as a process definition language. This approach leads to a very generic yet easily programmable process framework. A fully functional prototype implementation of the proposed framework is provided along with multiple example applications.

Despite the fact that business processes are increasingly automated and controlled by information systems, humans are still involved, directly or indirectly, in many of them. Thus, for process flows involving sensitive transactions, a highly secure authorization scheme supporting asynchronous multi-party transaction authorization must be available within process management systems. Therefore, along with the ePVM framework, this thesis presents a novel approach for secure remote multi-party transaction authentication - the zone trusted information channel (ZTIC). The ZTIC approach uniquely combines multiple desirable properties such as the highest level of security, ease-of-use, mobility, remote administration, and smooth integration with existing infrastructures into one device and method.

Extensively evaluating both, the ePVM framework and the ZTIC, this thesis shows that ePVM in combination with the ZTIC approach represents a unique and very powerful framework for building workflow systems and process-driven applications including support for secure multi-party transaction authorization.

1. INTRODUCTION

The subject of this thesis is the architectural design of a generic process engine framework along with a novel approach for secure remote multi-party transaction authorization. In this introductory chapter, the necessary background is provided by examining the recent past, evolution and the future trends of the two domains of process execution frameworks and secure remote authentication and transaction authorization. A discussion then evolves around these trends, leading to the motivation behind this project, its scope and goals, and the methodology adopted in order to design, architect, and implement a suitable solution. For such a solution to be realized, the first step is to identify the challenges related to architectural aspects as identified at the analysis and feasibility phases, and those related to technical aspects as emerged during the implementation and testing phases. These challenges are presented in this chapter together with a detailed requirements specification. Finally, the scientific contributions and the practical deliverables of this thesis are summarized, along with the possible application domains of the results.

1.1. WORKFLOW AND BUSINESS PROCESS MANAGEMENT

In computer science, the discipline of workflow management is well known since the 1980's. It has evolved from office automation research, which was focusing on reducing the complexity of office information systems and controlling the information flow within the office [1,2]. The Workflow Management Coalition (WfMC), founded in 1993, describes workflow management as

"the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules"[3]

The core goal is the automation of a business process with the help of an information system – the workflow management system (WfMS). A business process can be defined as a discrete, holistic, temporal and logical sequence of activities that are necessary to achieve a particular objective of an enterprise [4]. Furthermore, a business process can be coordinated via a WfMS, only if it is available in a specific well-defined representation, the so-called workflow definition or process definition. Throughout this thesis the two terms, workflow and process, are used interchangeably as in the context of this work they

have the same meaning. Process definitions can be loaded into a so-called execution engine, one or more process instances can be created from a given definition, and the execution engine then executes instances according to the definition. During execution, the workflows can interact with entities external to the workflow engine, for instance, a person or an application might trigger a workflow or vice versa. Additionally, a workflow engine typically provides monitoring functionality to trace workflow instances and their states.

To be able to establish standards, the WfMC has defined the workflow reference model, shown in Figure 1, which describes the basic components and interfaces of a WfMS [5].

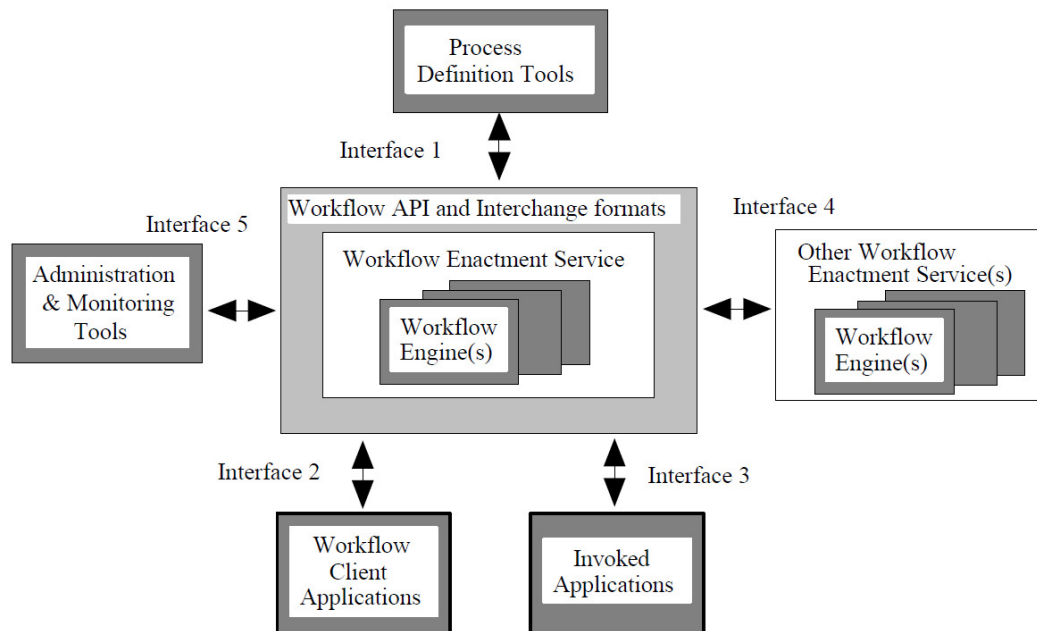


Figure 1. Workflow reference model – components & interfaces

Interface 1 is used to load process definitions into the workflow engine, sometimes called enactment engine, which resides at the heart of every WfMS. Interfaces 2 and 3 enable interaction with external entities whereas Interface 5 provides means for administration and monitoring. Finally, with Interface 4, the WfMC model also includes inter-engine communication.

Originally, workflow management was focusing on passing documents or tasks between people. However, with the increasing use of information systems within businesses, additionally linking various domain specific enterprise applications such as customer relationship management (CRM) or business intelligence applications into workflows

became more and more important. Therefore, workflow systems are frequently used in the so-called enterprise application integration (EAI) scenarios [6].

Nowadays, with the ubiquity of the Internet and the introduction of Web services (WS) [7] enterprises seek to unify their heterogeneous IT interfaces with the goal to establish a more open, more dynamic, and more manageable service oriented architecture (SOA) [8]. However, they realized that WS and SOA are not enough to solve the most important problem within companies – process management. To answer the question of how to choreograph and orchestrate the interaction between services in order to define higher-level business processes and how to continuously evaluate and optimize existing business processes, a new class of applications has emerged, the so-called business process management (BPM) [9]. BPM can be considered the successor or an extension of workflow management with a wider scope and, in the context of SOA, following a process-centric paradigm. BPM combines both, technical aspects such as workflow management and EAI as well as not necessarily technical aspects from the management domain such as business process optimization, quality management, or process reengineering [10]. Furthermore, BPM focuses on a comprehensive set of software tools, which allow for direct execution of business processes without the development of custom, domain specific software. This way, BPM aims to close the gap between business and IT people or between process modelling and execution, respectively. With the advent of BPM, many WfMS solutions have become, or have been renamed, BPM systems (BPMS) such that at the time of writing the term workflow management is often used interchangeably with BPM. IBM WebSphere Process Server [11] and JBOSS jBPM [12] are examples of commercial and open source BPM solutions, respectively.

Figure 2 shows the BPM lifecycle, which usually starts with the process modelling phase. BPM is influenced by the model-driven development (MDD) [13] paradigm in which high-level graphical software tools are employed to model a business process using graphical process notations. The tools then generate the corresponding textual process definitions, sometimes also called process interchange format or persistence format. A large amount of research and standards work is focusing on these two fields - graphical notations and process definition languages. Well-known examples of such standards are the Business Process Modelling Notation (BPMN) [14], the Unified Modelling Language (UML) [15], the Business Process Execution Language (BPEL) [16], and the XML Process Definition Language (XPDL) [17] as indicated in Figure 2. Process definitions are loaded into the process execution (enactment) engine, which

provides means for monitoring process instances during execution. Monitoring information can be used to generate audit trails for quality control as well as for process evaluation. Finally, the lifecycle is closed by feeding measures for improvement gained during process evaluation back into the design and modelling phase. This thesis focuses on the process execution part of the BPM lifecycle including process definitions and monitoring as its immediate input and output.

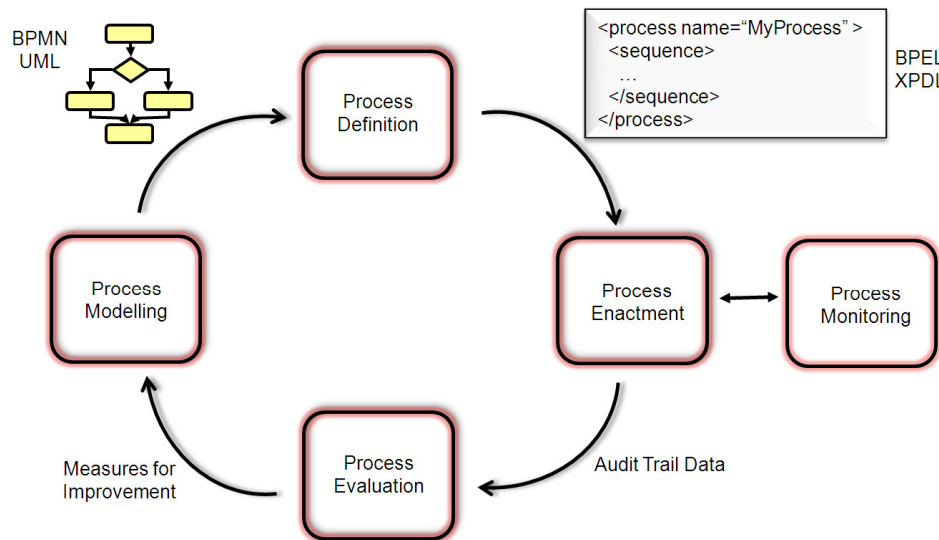


Figure 2. The BPM lifecycle

1.2. PROCESS-DRIVEN APPLICATIONS

Today's businesses are increasingly process driven. Ideally, all actions within an enterprise are explicitly defined as processes with the goal to improve control, flexibility, and effectiveness of delivering customer value. Additionally, business processes are oftentimes supported or even fully implemented by software applications [4]. In cases where various applications, services, and humans are involved to implement a business process BPM/WfM systems can be used to integrate and control process flows. Figure 3B illustrates such a scenario. However, in many cases the business processes are turned into a discrete piece of software, for instance, a domain specific enterprise application, such that they are hard-coded and hidden in the applications source code. To avoid this, there is a trend towards separating the main business logic from the functional code such that the resulting applications become more transparent and more flexible. The approach is to embed a process engine into the application, which then executes process definitions

representing the main control logic of the application. Application components providing the actual functional code are then triggered from the process engine in accordance with the process definition (c.f. Figure 3A). Such applications are called process-driven or workflow-driven applications [18,19]. The main advantages of this approach are the fact that the application logic can be modified without re-compiling the application, even at runtime, the business logic is more evident, and various features of the process engine, for example, monitoring or support for process persistence, can be exploited. Furthermore, using a process engine can ease dealing with concurrency and synchronization within applications.

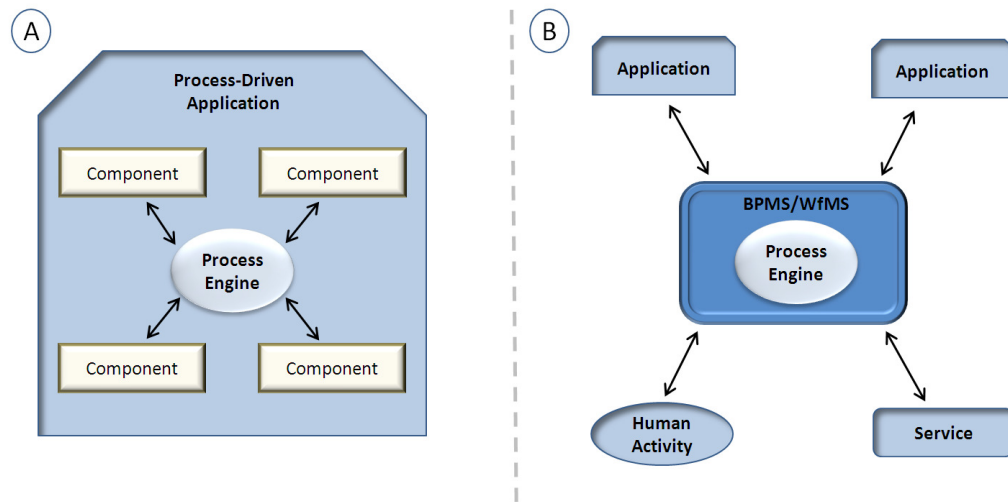


Figure 3. Process-driven applications vs. BPMS/WfMS

In the context of this thesis, a process-driven application is considered a software application, which is driven by an embedded process engine. ‘Embedded’ means that the engine is not a stand-alone discrete piece of software installed separately, like a WfMS, but is a library, which needs to be integrated. One could consider a WfMS a process-driven application, too. However, in most cases the functionality of the WfMS itself is not driven by process definitions executed by the process engine. Nevertheless, both require a process engine at its core as shown in Figure 3.

Many BPMS/WfMS are rather monolithic and domain specific such that they cannot be integrated into all kinds of applications. Additionally, their process engines are not reusable separately since they are not generic enough, have a lot of dependencies, or their interfaces are not well documented, if at all. Thus, one of the goals of this thesis is to define a generic framework for process execution, which can be used as the process engine for both application scenarios as illustrated in Figure 3.

1.3. SECURE REMOTE AUTHENTICATION AND TRANSACTION AUTHORIZATION

Despite the fact that business processes are increasingly automated and controlled by information systems, humans are still involved, directly or indirectly, in many of them. Actions of people often trigger the instantiation of processes in the first place. Additionally, people are involved when critical decisions are to be made during a process flow. For example, a purchasing process might be triggered by an employee who is entering an initial purchasing request into an information system. At a later point within the process, one or more people of the employee's management chain might be required to authorize the purchase such that the necessary funds become available. In such cases, it is important to provide means for securely authenticating people who make these decisions and to protect the decisions against fraudulent modification. This is what is meant by the term *secure transaction authorization* whereas a transaction can be any operation such as the authorization of a purchase, the allocation of a server resource, or granting of access rights, just to name a few examples. The level of security required depends on the level of confidentiality of the information to be dealt with. Financial information or trade secrets are typical examples of information that is considered most sensitive and thus demand the highest level of security.

The fact that nowadays people are working remotely while being connected with information systems via some potentially insecure network, be it the Internet or any other data network, adds another dimension to the challenge of authentication and secure transaction authorization. Internet banking is one of the prime examples where people interact with remote workflow systems processing highly sensitive financial transactions. Depending on the type of attack, it is not only the network connection, which is potentially insecure but also the client PC that is used to connect to a remote workflow system. Figure 4 illustrates the three main types of attack scenarios. Scenario 1 describes the so-called *phishing* attack, which usually consists of a combination of spoofed emails and mocked-up Web pages. An attacker might, for example, hijack a well-known institution's trusted brand and trick users into entering their credentials such as passwords or one-time codes into a faked Web form. Such trickery is commonly achieved through emails that look genuine if users only casually examine them. The infamous *man-in-the-middle* (MITM), shown in scenario 2 of Figure 4, is a network attack. Rather than trying to obtain a user's credentials, the attacker covertly intercepts messages between the client and server, masquerading as the server to the client and as

the client to server, respectively. Although connections are usually secured via protocols such as Secure Socket Layer (SSL) or Transport Layer Security (TLS) using public-key certificates, users often naively ignore warning messages about invalid or untrusted certificates. This lets attackers hijack an authenticated SSL/TLS channel or silently modify transaction data. Finally, scenario 3 of Figure 4 shows the so-called *malicious software* (MSW) attack, which aims to fraudulently gather the user's credentials by invading an insufficiently protected client PC by means of a virus or Trojan horse. For example, once established, a Trojan horse could read and forward a private key stored on the PC's hard drive while monitoring keyboard activity to access the pass phrase used to decrypt the private key. Users can protect themselves against malicious software using security precautions - such as installing and maintaining a firewall and regularly updating antivirus software; applying OS and browser patches as needed; and configuring software appropriately - but, apart from the fact that these precautions do not guarantee full protection, few users strictly adhere to such procedures. Besides gathering credentials, malicious software can also silently modify transactions done by the user turning the client PC into a completely untrusted device [20,21,22,23].

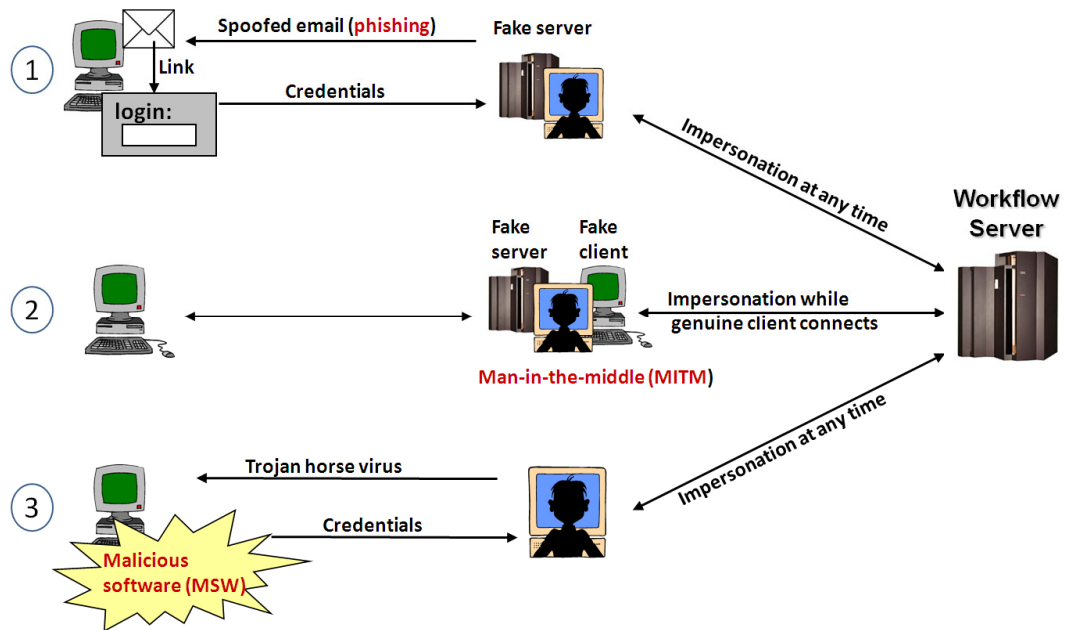


Figure 4. The three main types of attack scenarios

To counter these attack scenarios, a number of security schemes have been developed in recent years. Most of them follow the general approach of establishing an authenticated and encrypted information channel between the client PC and the server. Once

established, any transaction can be carried out over this channel without further authorization. To solve the problem of confidentiality and integrity, open standard protocols such as SSL or TLS have proven to be reliable and have become the de facto standard. However, for authentication, no single scheme has become predominant yet. Initially, simple static passwords, one-time codes (also known as scratch list), or so-called soft-tokens (keys stored on the hard drive in encrypted form) were used for client authentication. In many cases, they are still in use at the time of writing. Unfortunately, such credentials can be easily stolen via simple phishing attacks. Phishers reportedly convince up to 5 percent of users receiving a spoofed email to respond and reveal their secrets [24]. To prevent credential stealing, small handheld devices, so-called hardware tokens, which generate short-time passwords (valid for a limited time only) on demand, have been developed. These tokens successfully thwart credential-stealing attacks. However, recent research shows that due to their introduction attackers are increasingly moving to more advanced attacks such as MSW/MITM [25].

MSW attacks put the general approach of establishing an authenticated channel between the client PC and the server into question. Assuming that a virus has the client PC under control renders the secure channel, for instance, a TLS session, useless since the virus can control it as well. MSW effectively turns the client PC including display and keyboard into an untrusted device. Consequently, the user must have a trusted device, which does not only generate codes for authentication but also securely displays transaction information and supports secure input for explicit transaction authorization on the device itself.

At the time of writing, only a few solutions exist, which provide protection against MSW attacks. The AXS-Card [26], the mTAN solution [27], and the Secoder standard [28] are technically among the more promising ones. Yet, none of them combines an effective security concept with convenience, mobility, straightforward integration in existing infrastructures, and cost efficiency. Additionally, they are mostly designed for single persons to carry out online transactions as required in Internet banking, health, or government applications. In an industrial environment, however, multi-party transaction authorization is frequently required for various reasons such as organizational hierarchies or risk management. Thus, for sensitive transactions, a highly secure authorization scheme supporting asynchronous multi-party authorization must be available within process management systems. In this thesis, a novel approach for secure remote multi-

party transaction authentication is presented and integrated into the process engine as an optional security service.

1.4. AIM AND SCOPE OF THIS THESIS

The focus of this thesis is twofold. Firstly, it is on the architectural design of a generic process engine framework, which can be embedded into process-driven applications or used for building workflow and BPM systems. Secondly, a novel approach for secure remote multi-party transaction authorization is presented and integrated into the process engine framework. The main hypothesis of this research can be formulated as:

Workflow and BPM systems as well as process-driven applications can significantly benefit from a generic process engine, which is easily embeddable and independent of high-level, domain specific tools and process definition languages. Furthermore, process engines can profit from highly secure authorization methods, withstanding even malicious software attacks, for securing especially sensitive transactions. To achieve these goals, a suitable process engine architecture as well as a new secure transaction authorization method supporting multi-party authorizations are required.

This thesis provides a discussion and analysis of the scientific and technological issues involved in the design and validation of such architecture and authorization method. This includes identifying limitations of existing process engines and relevant process languages and tools, characterizing shortcomings of existing authentication and transaction authorization schemes with respect to MSW/MITM attacks, and presenting the techniques and implementation choices required to overcome these limitations and shortcomings. The scientific contributions that support the main hypothesis are:

- ❑ Detailed analysis, critical evaluation, and comparison of existing workflow and process engines with a focus on those that aim towards a generic and embeddable architecture;
- ❑ Detailed analysis, critical evaluation, and comparison of existing authentication and transaction authorization methods, which potentially thwart MSW/MITM attacks;
- ❑ An architecture for a generic and embeddable process virtual machine (ePVM), which supports defining process flows along the theoretical foundation of communicating extended finite state machines (CEFSM) and focuses on core process functionality such as control flow and state management, monitoring, persistence, and communication;

- A persistence model for state-machine based process engines, which leads to improved flexibility in defining persistence behaviour and thus enables optimization of performance and resource consumption during process execution;
- The presentation of a new secure transaction authorization method and apparatus, suitable for mobile multi-party transaction authorization, and its integration into the process engine;
- A fully functional prototype implementation of the proposed process engine framework including support for secure multi-party transaction authorization;
- Evaluation of the process engine and transaction authorization method prototypes via a series of experiments and a comprehensive process-driven use case application focusing on large-scale biometric identification.

1.5. MOTIVATION

The motivation for this thesis project has evolved from another project for which a number of workflow engines have been informally evaluated. The goal was to find an engine, which could be integrated into a customer relationship management application such that the main application logic could be centrally controlled and monitored. In the course of this evaluation, the following general issues have been identified:

- A large number of rather domain specific process definition languages exist but there is no convergence towards a dominating one.
- Most process engines are built for one particular process definition language and are therefore domain specific, too.
- Most process languages are XML based declarative languages, which are not easily programmable by hand.
- Process engines are often heavyweight, monolithic systems with many dependencies, for instance, based on application servers, and thus not embeddable into arbitrary applications.
- In accordance with the MDD paradigm, many frameworks rely on high-level tool support such as graphical modelling and code generation.

Therefore, the motivation behind this thesis project is to make a step away from domain specific monolithic workflow engines towards generic and versatile process runtime frameworks, which enable integration of process technology into all kinds of software. To achieve this, the idea of ePVM as a lightweight and generic core engine following a bottom-up or micro-kernel type of approach has been developed. Such a generic and

easily programmable engine could then be embedded into all kinds of process-driven applications. Alternatively, one or more domain specific process languages and related high-level tools could be supported on top of it with the goal to better support evolution of such languages or to build fully-fledged WfMS/BPMS.

An additional aspect of this thesis project is to address the issue of secure remote authentication and transaction authorization. Nowadays, people interacting with process-driven applications or workflow systems usually do this remotely via potentially insecure networks such as the Internet. There are a number of methods to protect such interactions, for instance, protocols such as SSL/TLS and security tokens such as one-time passwords or smart cards. However, most of them only provide protection against phishing attacks or network attacks like the infamous man-in-the-middle. Only a few methods consider situations where the client PC used to connect to the remote workflow application is under attack by malicious software. These methods, if effective at all, have shortcomings with respect to various other properties such as convenience, mobility, interoperability, administration, or cost. A new approach not exhibiting these shortcomings while providing the highest level of security would significantly improve the applicability of remote interactions, in particular, for highly sensitive applications. Internet banking is a good example for such an application. Providing the users with means to securely verify and authorize transactions would increase trust in electronic business in general. Furthermore, such a method would build the ground for secure multi-party transaction authorization, as it is frequently required in business processes. Integrating the security aspect with the generic process engine would result in a very versatile process execution framework, which could be applied in all kinds of application domains including those dealing with highly sensitive information.

1.6. METHODOLOGY

This thesis presents the motivation, analysis, architectural design, prototype implementation, and use-case based evaluation of the ePVM framework. In the same way, except for the implementation, a new approach for secure remote multi-party transaction authorization and its integration in ePVM is presented. The analysis is based on a set of requirements extracted from the challenges that the objectives of this thesis impose, and the lack of an existing solution that fully meets these requirements as resulted from the related work review. The architectural design of the process engine and its process model is based on the theoretical framework of communicating extended

finite state machines (CEFSM) [29]. The thesis transports the ideas of CEFSM into the context of process execution frameworks and turns them into a practical programming model and runtime framework architecture. The prototype implementation of ePVM uses Rhino [30] – an open source JavaScript interpreter written in Java. This ensures the highest degree of interoperability and portability. The evaluation phase was supported by a process-driven use case application, built in the context of the GridCOMP European research project [31], which was used to carry out a case study.

1.7. APPLICATION DOMAIN

The process engine presented in this thesis along with the transaction authorization method represents a generic framework, which can potentially be applied in virtually all conceivable applications. Three main application classes that can particularly benefit from the results of this thesis are:

- **Business and enterprise applications:** Here, it is especially important to have central process control and support for real-time monitoring. Furthermore, enterprise applications typically involve many concurrent instances of pre-defined process flows and businesses increasingly ask for flexible software to keep up with rapid market change. This is the prime environment for process-driven applications.
- **WfMS/BPMS:** These systems can benefit from a generic and clearly separated core process engine in two main respects. Firstly, they can better support evolution and interoperability as new/additional standard process definition languages can be added on top of the generic core engine. Secondly, they become less dependent on high-level tools and languages, as the core engine is independent from those. Users appreciate this, as they usually want to avoid hard dependencies on such tools for various reasons, for instance, incompatibilities among different versions, no option to optimize or work around issues by hand, and so forth.
- **Highly sensitive applications:** Applications dealing with highly sensitive data do not only require process support but also support for secure authentication and transaction authorization. Such applications, which might well be rooted in the previous two application domains, can benefit most from the results of this thesis, as they can profit from all results presented at the same time. What is considered highly sensitive is hard to define, depends on individual perception, and changes

over time. Typical examples come from the financial, government, health, or military domains. However, other applications such as remote system management could well be considered very sensitive, too.

Concrete examples of such applications may include (but are of course not limited to):

Order management

Order management is the most common example application used in the BPM/workflow field [16]. Typically, a large number of orders exist concurrently and it takes rather long until an order is processed completely. Each order process involves interaction with and synchronization of certain activities such as production, payment, and shipment. This is a typical application scenario where the use of a process engine makes a lot of sense. A generic and versatile process engine such as ePVM additionally increases the possibilities of how to implement the application.

Revision control system

A revision control system (RCS) for managing an application's source code is not a typical example where using a process engine comes into one's mind immediately. Therefore, process flows in such applications tend to be hard coded and interweaved with the applications source code. However, considering the functionality of a RCS more closely unveils that it is an ideal candidate for a process-driven application, as its functionality largely is coordinating source code related rules and activity flows. Clearly separating these flow definitions from the functional code would improve the RCS in terms of maintainability and flexibility. A process engine, which is embeddable and easy to program by hand significantly simplifies the effort of building a process-driven RCS.

BPMS

As described in Section 1.1, a process engine is located at the heart of every BPMS/WfMS. Consequently, the properties of the core engine used are very important when building such a system, as it has an impact on the higher-level process languages and modelling tools to be implemented on top. A generic core engine obviously improves flexibility and supports evolution of the overall system, for example, if support for additional process languages or tools must be added later on.

Online banking

Depending on the detailed application scenario concerning security requirements, a strong authentication and transaction authorization method might or might not be necessary in the examples described previously. For every online banking application, in contrast, it definitely is an inherent requirement. At the same time, the backend system

must implement certain banking related process flows. Consequently, all results of this thesis can be exploited when designing and building an online banking application.

The biometric identification system presented in Chapter 1 represents an additional real-world example, which exploits all results of this thesis project.

1.8. CHALLENGES AND REQUIREMENTS

This section points out the challenges that the goals of this thesis project impose and from which the relevant requirements can be deduced.

1.8.1. Process Execution Framework Challenges

1.8.1.1. Interoperability

To turn the BPM idea into reality a number of process definition languages have been created in recent years, with BPEL being the most prominent example. The same is true for graphical notations used in process modelling. At the time of writing, BPMN seems to become increasingly popular as with the new version of the standard (version 2.0) it not only defines the graphical notation but also the corresponding process interchange format – yet another one. Overall, the standards landscape includes more than seven languages and over ten standardization groups with interest in the field [32,33,34]. Despite many standardization efforts, there seems to be almost no convergence towards a dominating standard. One reason for having such a diverse set of languages and notations is the fact that process technology is applicable to all kinds of application domains. As a result, languages and related execution engines are rather domain specific and lack interoperability, which prevents broad adoption. For example, BPEL has been developed in the context of SOA and is by-design a Web services based language. This means that all entities a BPEL process interacts with must be available as a Web service and defined via the Web Service Description Language (WSDL). Obviously, introducing Web services in order to be able to use BPEL does not necessarily make sense in all application environments. Nevertheless, in many projects it has been done just because BPEL centric execution engines were popular and available at that time.

Although there is an issue with interoperability, it is likely that the number of domain specific languages will keep growing and different languages will coexist. From a domain specific perspective, this actually makes sense since some languages are simply better suited for certain applications than others. However, most attempts to solve the

interoperability problem so far usually result in proposing another, supposedly more powerful, process language or notation and expecting it to become the dominating standard [35]. Additionally, execution engines are then built specifically for a particular domain specific language. This thesis aims to adopt a different approach. Firstly, a process execution system should be based on a generic core engine framework on top of which support for one or more domain-specific process definition languages can be built. This way, the resulting system can outlast rapid standards change and evolution. The idea is to follow a bottom-up or micro-kernel type of approach to build a generic execution engine, which only provides features common to all process languages and execution systems. Secondly, it should not introduce any new imperative, declarative, or graphical process language but in contrast provide a generic runtime framework and programming model where higher-level domain specific languages can be mapped to. Yet the framework should be self contained and directly usable without such mappings. Only in recent years, a few research and development efforts into this direction have appeared [36,37,38,19,39].

1.8.1.2. Versatility

Besides interoperability, versatility is an equally or even more important property of a process execution framework. Of course, having a generic framework to some extent also contributes to versatility but there are additional aspects to be considered. At the time of writing, a trend towards process-driven application development (c.f. Section 1.2) can be observed [40]. This means that all kinds of applications, that is, not only traditional document-centric workflow applications or BPM systems, can make use of a process execution engine to implement their core logic. The goal here is to make the resulting software more adaptable to the rapid changes required by today's business environment. However, BPM systems are often heavyweight, monolithic systems with many dependencies imposing a number of requirements on their environment. For instance, some execution engines rely on a particular application server technology such as enterprise Java (J2EE) [41], some work on high-level standards such as Web services exclusively, and others may require a particular database system to be available. As a result, these engines hardly scale in size and functionality and thus are not easily embeddable into diverse application scenarios. Therefore, a process execution framework should not only be generic but also lightweight and embeddable. Embeddable means that it includes an appropriate application programming interface (API), which allows

programmers to easily embed the process engine into another application, the so-called host application. Furthermore, there must be an interface for attaching arbitrary functional code to the engine in a way such that processes can interact with this code. For instance, for attaching business code to be triggered or adding higher-level protocol support such as Web services communication. Lightweight means that the execution framework is reasonably small in terms of code size and that it imposes minimum requirements on its environment, namely the host application. In other words, using it should not necessarily require shipping/installing complex software packages such as Web servers, database systems, or other third-party runtime environments. These are important requirements making it suitable for embedding not only in BPM systems but also in diverse process-driven applications.

1.8.1.3. Programmability

Following the trend towards declarative programming, most state-of-the-art process definition languages are based on XML. XML, however, is not easily programmable by hand because its encoding is very ‘noisy’ and, although working well for defining state, it is not particularly well-suited to define behaviour. Consequently, XML-based languages seldom support fine grained procedural logic and complex data manipulation. Listing 1 illustrates what is meant by ‘noisy’ encoding and not easily programmable by hand. It shows a code snippet of an activity rule as it is generated by the visual workflow editor for Windows Workflow Foundation [19]. It defines an expression of a condition representing a workflow activity, which means a step within a sequential workflow. Studying this XML code for quite a while unveils that it only defines the simple expression “**amount < 1000**”. Obviously, such encodings are only suitable for use as a persistent storage or process interchange format, respectively. Reading or writing such XML code manually is cumbersome.

```
...
<ns0:RuleConditionCollection
  xmlns:ns0="System_Workflow_Activities_Rules">
  <ns0:RuleExpressionCondition Name="AutoApproveCondition">
    <ns0:RuleExpressionCondition.Expression>
      <ns1:CodeBinaryOperatorExpression Operator="LessThan"
        xmlns:ns1="System_CodeDom">
        <ns1:CodeBinaryOperatorExpression.Left>
          <ns1:CodeFieldReferenceExpression FieldName="amount">
            <ns1:CodeFieldReferenceExpression.TargetObject>
              <ns1:CodeFieldReferenceExpression />
            </ns1:CodeFieldReferenceExpression.TargetObject>

```

```

</ns1:CodeFieldReferenceExpression>
</ns1:CodeBinaryOperatorExpression.Left>
<ns1:CodeBinaryOperatorExpression.Right>
  <ns1:CodePrimitiveExpression>
    <ns1:CodePrimitiveExpression.Value>
      <ns2:Int32 xmlns:ns2="System">1000</ns2:Int32>
    </ns1:CodePrimitiveExpression.Value>
  </ns1:CodePrimitiveExpression>
</ns1:CodeBinaryOperatorExpression.Right>
</ns1:CodeBinaryOperatorExpression>
</ns0:RuleExpressionCondition.Expression>
</ns0:RuleExpressionCondition>
</ns0:RuleConditionCollection>
...

```

Listing 1. Windows workflow foundation activity rule snippet

To enhance programmability, some process languages allow mixing XML code with imperative programming languages. For instance, BPELJ [42] allows embedding snippets of Java source code into the XML structures. However, this only solves part of the problem at the cost of further diminished readability. To ease programming, some systems are fundamentally based on graphical modelling and code generation tools in the spirit of the MDD paradigm, an approach that is very good and works quite well in various scenarios. However, there are cases where an option for more traditional programming should be available, for instance, to overcome tool limitations and bugs or for optimization purposes. Thus, the core execution engine should be independent from high-level tools and languages to allow for an intermediate point of control or standalone deployments, respectively. Most of today's execution engines either do not provide such an intermediate-level interface or it is kept internal and/or not well documented.

Despite the fact that the execution engine represents the heart of every BPM or workflow system, it is rarely discussed how high-level definitions are mapped to code executable on contemporary runtime systems. Only a few process engines explicitly provide a low-level programming interface [43,40]. Additionally, these interfaces are usually provided in form of a graph oriented object or component model. Graph Oriented Programming (GOP) represents a strategy for implementing graph execution on top of an object-oriented programming language [44]. In other words, it provides means to structure software around graphs. The advantage of this approach is that there is always a graphical representation available for every process definition. The disadvantage is that a process definition using an object oriented programming language in combination with a particular graph oriented object model is rather complicated. Nevertheless, all process execution systems are based on the assumption that for every process there must always

be an explicit link to a corresponding graphical representation. For typical workflow and BPM systems and their applications, this assumption is valid. However, considering process-driven applications, this assumption does not hold. For instance, it might well make sense to use a process engine within a revision control system application without requiring a graphical process representation, in particular, if the process definitions are easily programmable by hand. Therefore, a generic process execution framework should be easily programmable by hand using an easy and well-known programming language, similar like in scripting environments. Support for graphical representation or higher-level languages should be optional.

1.8.1.4. Definitions Summary

The definitions of various terms made in this chapter are summarized below. These terms are frequently used throughout the thesis in connection with process execution engines.

Lightweight: A lightweight execution engine is reasonably small in terms of code size and it imposes minimum requirements on its environment. For instance, it consists of a library of a few megabytes and it does not require complex software packages such as Web servers, database systems, or other third-party runtime environments.

Generic: A generic process engine only provides core features common to all process languages and execution systems, for example, state and control flow management, communication, monitoring, and persistence. Higher-level domain specific functionality such as graphical modelling or support for Web services are not directly supported.

Embeddable: An embeddable process engine is specifically designed to be integrated with arbitrary applications, for instance, by means of an appropriate integration API. It is not an off-the-shelf application that can run standalone.

Versatile: A process engine is versatile if it is lightweight, generic, and embeddable.

Interoperable: An interoperable process engine can potentially support more than one process definition language. Additional languages can be added dynamically to support the evolution of process languages.

Programmable: A programmable process engine makes it possible to create process definitions by hand using an easy and well-known imperative programming language similar like in scripting environments. Creating process definitions does not require high-level tools for graphical modelling and code generation.

1.8.2. Secure Remote Multi-Party Transaction Authorization Challenges

This section describes a broad range of challenges going beyond the technical and security related aspects of transaction authorization. Additional aspects such as ease-of-use, mobility, administration, and integration are also considered, since they are of equal importance for real-world deployments. There are solutions covering certain subsets of the given challenges. Therefore, the ultimate challenge is to define a method that satisfies all of the challenges and requirements discussed in this section at the same time. Figure 5 graphically summarizes the main challenges to be addressed with security at its core.

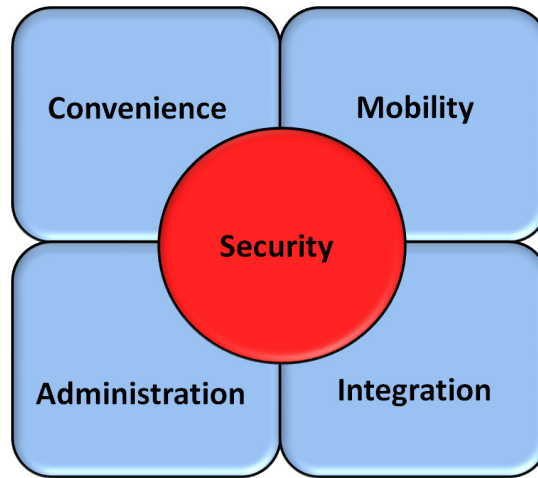


Figure 5. Transaction authentication challenges

1.8.2.1. Effectiveness against Software Attacks

Obviously, the foremost challenge is to thwart all types of attacks illustrated in Figure 4 including all kinds of combinations thereof. Thus, the core requirement for a secure transaction authorization method is to be effective against these attacks, namely phishing, man-in-the-middle, and malicious software. It is to be noted that these attack types only cover software attacks, and with phishing to some extent social attacks, while physical attacks are not considered. The reason is that with physical attacks only one or a few users can be attacked, at least with manageable effort and cost. Also, physical attacks can be investigated and traced using standard criminological methods. In contrary, with software attacks it is very easy to launch mass attacks, for instance, by spreading a Trojan horse, and it is hard to trace them back. At the time of writing, the threat by malicious software is significantly increasing and it is the one that worries people and institutions most [25].

In addition to these fundamental security requirements, a transaction authorization method should provide means for non-repudiation and qualify for multi-party usage. The latter is particularly important in industrial applications where transactions need to be authorized by multiple people. The former is important in cases where authorizations must be legally binding. Non-repudiation is usually achieved by digitally signing transactions using hardware tokens such as chip cards.

The majority of methods in use today do not satisfy the requirements discussed above. For example, the usage of one-time or short-time passwords in online banking applications is still prevalent, even in Europe, which is compared to other parts of the world quite progressive with regards to security and privacy. Nevertheless, there are a few methods providing protection against software attacks, for example, solutions based on the German HBCI Secoder standard [45] or the standards developed within the Financial transactional IC card reader (Finread) EU project [46]. The reason why they never became widely accepted is the fact that they lack one or more of the other important properties discussed in the following subsections. Transaction authorization methods used in other industries tend to be even weaker than those applied in the financial world.

1.8.2.2. Ease-of-use and Mobility

Looking at current transaction authorization methods immediately shows that combining the highest-level of security with convenience and ease-of-use is challenging. Many are very complex and time consuming to operate. The so-called PINsentry system Barclays is currently offering their online banking customers is a good example illustrating this [47]. Users do not only need to carry a bulky smart card reader device but also have to enter all kinds of information via its keypad and manually copy passwords back from the device to the PC.

Besides ease-of-use, mobility is another challenge for a transaction authorization method. In today's dynamic and mobile world, it is important for users to be able to work anywhere and anytime. To support mobility, a transaction authorization method must fulfil three main requirements. Firstly, the security device required to carry out transactions should be easy to take along. In other words, it should have a small form factor, for instance, a key fob type of device. Secondly, if the device is to be connected to a PC, it should not require any software installation. Here, the goal is to avoid trouble during installation and to avoid spreading the software. Finally, if the device is to be

connected to a PC, it should be compatible with all major types of hardware, operating systems, and Web browsers (e.g. Windows, Linux, MAC OS, Safari, IE, Firefox etc.). Furthermore, the mobility requirement is especially important to enable efficient multi-party transaction authorization. If multiple people are involved in an authorization process, some of them might be travelling and thus must be able to work with pending transactions while being on the way.

Evaluating methods currently in use also shows that approaches where security devices do not need to be connected, like PINsentry, tend to be mobile but not easy to use. Others, using connected devices are rather convenient but not mobile, for instance, Secoder devices. Overall, there seems to be no single method, which combines ease-of-use and mobility without sacrificing security.

1.8.2.3. Ease-of-integration and administration

For a transaction authorization method to be generally applicable in real-world scenarios, it is crucial that it integrates well into existing environments. On the client side, this means that it must be able to interact with existing software and hardware without installing additional software as mentioned in Section 1.8.2.2. Most existing solutions using connected devices, for example, Secoder or Finread devices require special device drivers to be installed. Furthermore, it should be possible to reuse existing user credentials such as passwords or chip cards in order to retain existing investments. On the server side, ease-of-integration means that minimal server-changes are required. Ideally, existing authentication and transaction authorization protocols can be reused. In addition, cost efficiency is another relevant challenge. Here, the requirement is not to exceed the cost of current methods. Not considering cost would make the challenge very easy, as one could simply use a dedicated PC with limited functionality only used for one well-defined application such that the risk of being infected with malicious software is significantly reduced.

Finally yet importantly, administration of secure devices used in transaction authorization systems must be considered. It is important that such devices can be (re-)configured remotely in a secure way and without user interaction. This way, they might be personalized, certain risk settings might be adapted, or software might be updated. A fast and secure mechanism for doing this is especially important if security relevant bugs must be fixed.

1.9. DIRECTION OF RESEARCH AND THESIS ORGANIZATION

Logically the first step in this research effort is to study and critically analyse the challenges and requirements that a generic process execution framework poses. The focus is on laying the foundation for interoperability, versatility, and programmability. In addition, the requirements for a remote authentication and transaction authorization system, which not only provides the highest level of security but also addresses other important properties, are discussed. These properties include ease-of-use, mobility, integration, and administration often neglected by contemporary solutions.

With the requirements specification clearly identified, the next logical step is to examine existing technologies and related projects that provide process execution frameworks and transaction authorization methods, and identify their limitations in the light of the specified requirements (Chapter 2). Parts of this critical examination of related projects and solutions have been published in [48,20,21].

After studying existing approaches in both categories of interest, two important conclusions were reached: first, that there is no readily available and easily embeddable generic process engine that satisfies the given requirements. Second, none of the existing authentication and transaction authorization methods incorporates all the desired properties. Therefore, in order to meet the specified requirements, a generic framework for process execution is presented in Chapter 3. The framework, named the embeddable process virtual machine (ePVM), is discussed in detail along with its prototype implementation and an initial example application. Chapter 3 is based on work published in [49,48].

Furthermore, in Chapter 4, a novel approach to secure multi-party transaction authorization - the zone trusted information channel - is presented. Additionally, it is shown how the approach has been integrated into the ePVM framework to adequately secure highly sensitive transactions oftentimes carried out as part of business processes. Chapter 4 is based on work published in [50,20] and several related patents (c.f. Section *Declaration*).

The final step in this research effort was the extensive evaluation of the process execution framework and the transaction authorization method. Firstly, they were evaluated separately by means of a series of experiments. Secondly, they were evaluated in the context of a comprehensive process-driven application built as part of the GridCOMP EU project. Chapter 5 presents the experiments and the use case

application along with the gained results, experiences, and conclusions based on work published in [51,49,52,53,54].

2. RELATED WORK

The introduction of the necessary background concepts in Chapter 1, now allows the study and analysis of work done in the field of process execution frameworks and authentication and transaction authorization methods. Firstly, a synopsis is presented, that includes a diverse set of well-known process frameworks coming from the industry, open source, and academic domains. Each framework is briefly introduced and discussed with respect to the challenges and requirements presented in Section 1.8.1.

Secondly, an overview of state-of-the-art remote authentication schemes and the available client security devices used by them is provided followed by a treat-solution taxonomy relating the schemes to the relevant attack scenarios. Finally, a number of solutions that potentially provide protection against MSW attacks and are thus particular relevant for this thesis are presented and discussed with respect to the challenges and requirements set out in Section 1.8.2.

2.1. PROCESS EXECUTION FRAMEWORKS

2.1.1. Introduction

There are a very large number of diverse process frameworks available from the industry, open source, and academic domain. An overview can be found in [55]. This subsection introduces some of the frameworks relevant for this thesis project. Each approach is briefly introduced and discussed with respect to the challenges and requirements set out in Section 1.8.1.

2.1.2. IBM WebSphere Process Server

Many of the commercial BPM systems from big industrial players such as IBM, SAP, or Oracle are holistic and rather monolithic solutions following the SOA and MDD paradigms. IBM WebSphere Process Server (WPS) is a typical representative from this application space [56]. NetWeaver [57] and Oracle BPM (formerly known as BEA AquaLogic) [58] are the corresponding solutions from SAP and Oracle. Holistic means that these solutions comprise several applications, tools, and methodologies covering the complete BPM lifecycle. As shown in Figure 6, WPS is just one out of many products belonging to the WebSphere BPM family. WebSphere Business Modeler, for instance, is the graphical process-modelling tool, which supports BPEL and BPMN.

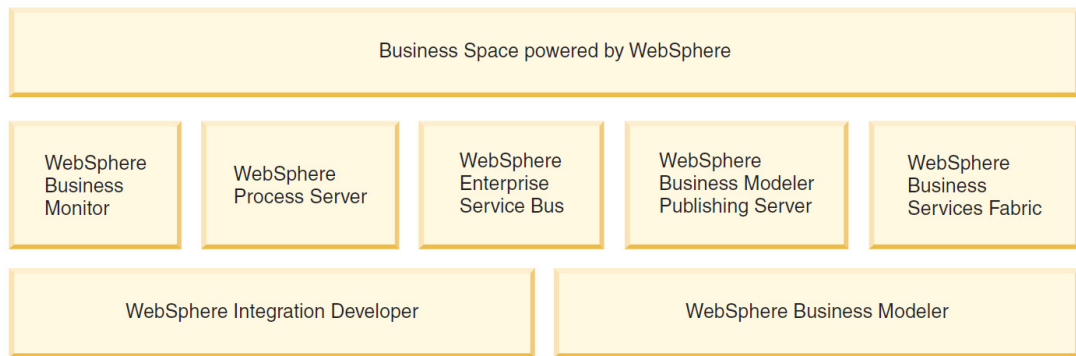


Figure 6. WebSphere BPM products

Taking a closer look on WPS, illustrated in Figure 7, unveils that WPS again consists of a layered architecture of building blocks all implemented within WebSphere Application Server (WAS) – IBM’s J2EE implementation. The actual process execution engine hides behind the *Business Processes* service component shown in Figure 7 and supports BPEL process definitions including a number of IBM extensions.

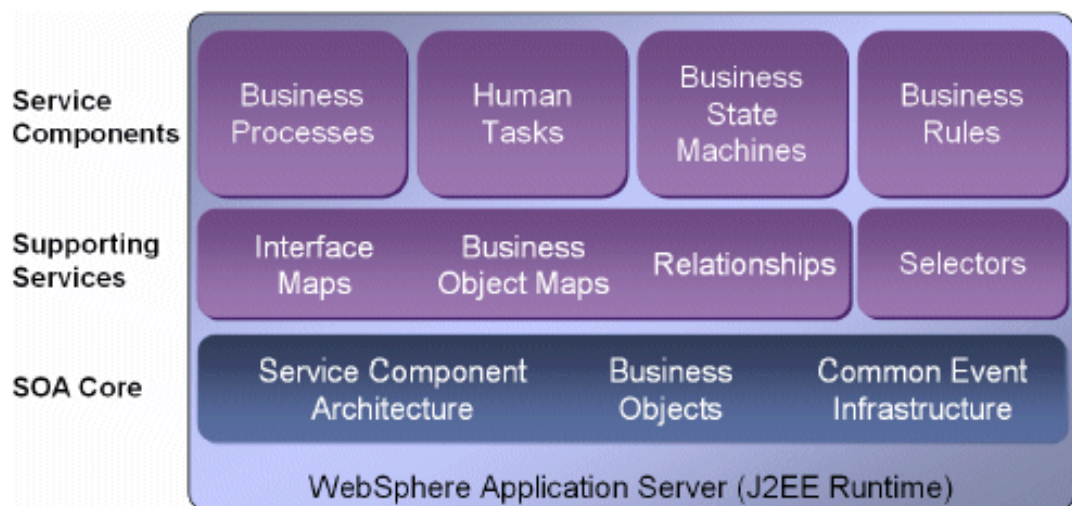


Figure 7. WebSphere process server platform overview

Gathering some hands-on experience with WPS and WebSphere Business Modeller by taking a one-week introductory course showed that all these software components are powerful and comprehensive but they are also heavily interweaved and not usable standalone. With regard to the requirements discussed in Section 1.8.1, the following can be noted:

- The process engine is rather domain specific, as it is based on BPEL and designed to operate in a SOA and Service Component Architecture (SCA) [59] environment.

- The process engine is integrated with all the other WPS components and can only be run within WAS. WAS, in turn, requires a large number of additional software such as IBM DB2 or IBM MQSeries to be installed in order to host WPS. This means the process engine cannot be embedded into arbitrary applications.
- It is assumed that process definitions are available as BPEL definitions and that they are generated via the graphical modeller. The definitions are automatically mapped to a proprietary J2EE object model at deployment time. Thus, process definitions can hardly be programmed by hand.

Obviously, IBM WPS does not satisfy the requirements set out within this thesis. The same is true for many holistic BPM solutions from other companies. These solutions rather gave rise for carrying out this project, as they induced the desire for a generic, embeddable, and easily programmable process framework.

2.1.3. Windows Workflow Foundation

Microsoft's Windows Workflow Foundation (WF) belongs to the recent research and development efforts that are very relevant for this thesis. About a decade ago, research work towards lightweight, flexible, and generic workflow kernels was carried out already. For instance, the Micro Workflow [60], Mentor-lite [61], and OPERA [62] projects were targeted into this direction. However, with the hype around SOA, MDD, and BPM, the focus apparently was shifted to higher-level process languages and graphical tools. Microsoft brought this topic back onto the agenda by introducing WF as part of the new Windows Vista operating system and the .NET 3.0 framework (formerly known as WinFX) [40,19]. People from academia [63] as well as from the open source domain (c.f. Section 2.1.4) also realized that these ideas should be picked up again.

What makes WF relevant for this thesis project is the fact that it consists of an embeddable and to some extent generic process engine framework. As illustrated in Figure 8, the WF components sit on top of the .NET runtime, which makes the framework somewhat platform dependent. Its workflow functionality is rather generic and not bound to any specific process language such as BPEL or XPDLL. Essentially, WF is nothing but an extensive class library that constitutes a workflow framework. Workflows are defined by creating a corresponding object model using the given class library. For example, in WF, an *Activity* represents a step in a workflow and it is the fundamental building block within WF. Thus, for each workflow step a corresponding activity object, which has to be derived from a specific WF base class must be created.

Such activities can then be arranged to form certain control flows using so called flow control activity objects, and so forth. Such workflows can then be instantiated and executed via the WF runtime engine. Actually, the runtime engine is not a separate application, but it is also represented by a class of the WF class library. This means, an instance of this class must be created and hosted by an application, the so-called host application, in order to execute and manage workflows. In other words, an application hosts the workflow runtime and the runtime hosts the individual workflow instances. Furthermore, the runtime engine supports the concept of external runtime services (c.f. Figure 8), which come in two different flavours: core and local. External means that these services are pluggable with respect to the runtime engine. Core services implement functionalities typical for process engines and default/example implementations are included in WF. The WF core runtime services are:

1. **Scheduling:** Creates and manages threads used to execute workflows
2. **Commit Work Batch:** Manages transactions used by the runtime engine
3. **Persistence:** Handles persistence of a workflow instance at the direction of the runtime engine
4. **Tracking:** Provides the ability to instrument workflow instances by recording tracking events

Local runtime services, sometimes called data exchange services, are optional services acting as a conduit for communication between a workflow instance and the host application. One or more local services can be registered with the runtime engine. Each local service provides methods and events that can be used by a workflow instance. For instance, a workflow instance can directly call a method of a registered local service in order to trigger some external business code. Such method calls are synchronous running in the thread of the workflow. In the opposite direction, the host application can send asynchronous events to a workflow instance. Such events are put into the event queue of the workflow and picked up by the workflow at some later point in time such that the event processing takes place within the workflow thread, not within the thread used for raising the events.

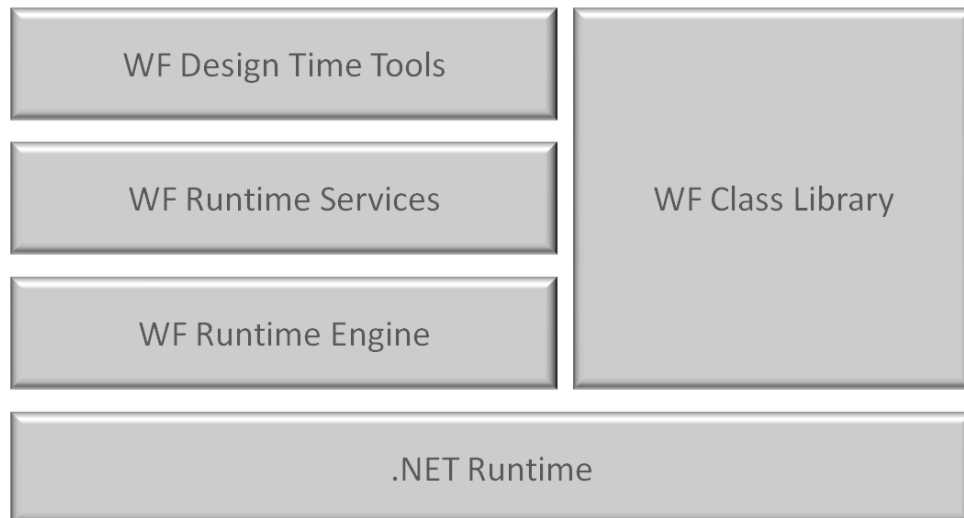


Figure 8. Workflow foundation components

Finally, WF includes a set of design time tools (c.f. Figure 8) mainly consisting of graphical workflow design capabilities integrated into Microsoft's Visual Studio development environment. These tools support visual construction of workflow graphs by placing and wiring activity objects and generating the corresponding object model automatically. The tools can be considered optional as the WF class library and runtime engine can be programmed directly (via the WF API). WF supports two different types of workflows: sequential and state machine. Sequential workflows declare a series of steps that are executed in a predefined order and they correspond to flow charts. Figure 9 shows an example of a sequential workflow designed using the graphical workflow designer of Visual Studio. It defines a simple expense approval process, which handles incoming expense request events and decides if the request should be approved or not in accordance with the WF rule shown in Listing 1 (c.f. Section 1.8.1.3). Depending on the result of the rule evaluation, the according method of a local service is called to trigger relevant business code.

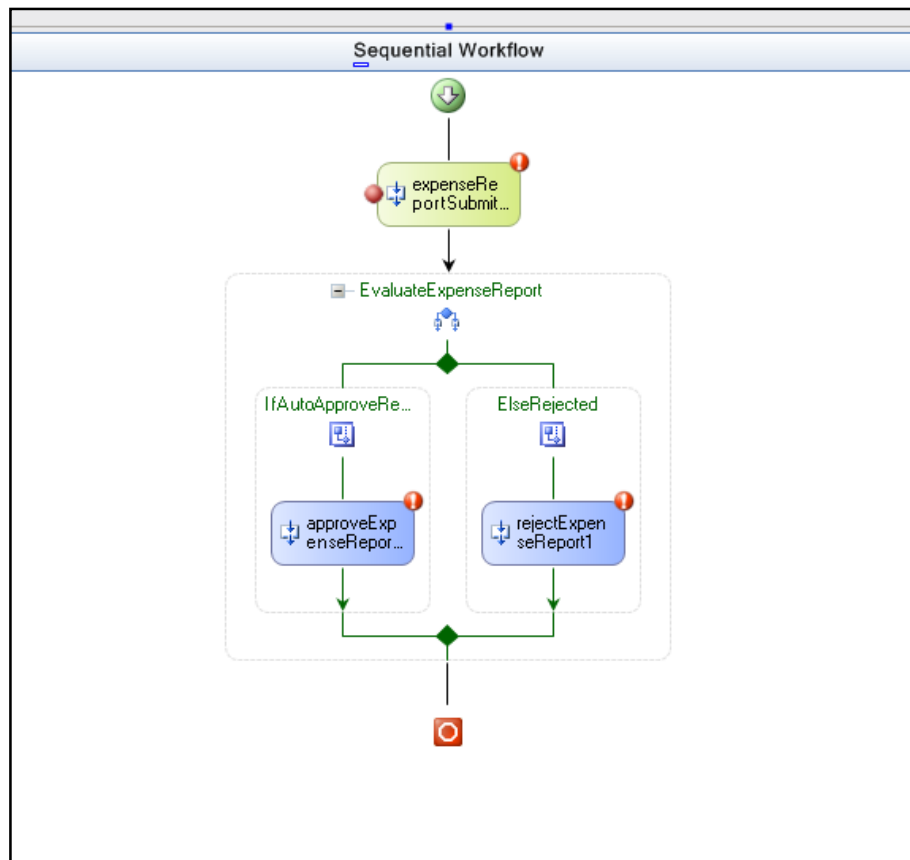


Figure 9. A sequential workflow example drawn in Visual Studio

Sequential workflows define a fixed flow of control meaning a fixed sequence of steps within the workflow. Thus, they are applicable whenever the prescribed steps of a workflow are clearly known at design time. However, more complex and dynamic business processes possibly involving human interaction require a more flexible way of process modelling. For such cases, WF supports state machine workflows. State machines do not define a fixed flow of control. They only define a set of states with possible transitions between the states. The exact sequence of state transitions is then dynamically controlled by external events at runtime. Users can choose which type of workflow modelling they want to use.

As mentioned earlier and shown in Listing 1 (c.f. Section 1.8.1.3), workflows in WF can include rules that are evaluated during workflow execution. WF defines a declarative rule language and includes a corresponding rules engine. Although code activities can implement the same functionality as rule activities, having a dedicated rule subsystem seems to be popular in workflow frameworks. The reason apparently is that the rule

languages are easier than, for instance, C# or Visual Basic code, and that the rules are interpreted at runtime and thus do not require recompilation.

WF supports three different ways to authoring workflows: code-only, code-separation, and no code. Code-only means that workflow developers work directly with the WF object model using the WF class library. The complete code, normally written in C# or Visual Basic, is then compiled as a class in a .NET assembly. Listing 2 exemplifies what such code looks like. For every step of the workflow, a corresponding activity object must be created and the object must be wired with respect to the desired workflow graph. Even simple if-statements, as known from high-level programming languages, must be turned into *IfElse* activity objects. The advantage is that the object model directly reflects the workflow graph, which means a graphical representation can be easily generated from the object model. However, programming workflows that way is rather cumbersome compared to programming the desired logic straight in the programming language.

```

namespace SequentialWorkflowWithParameters
{
    public partial class BasicSequentialWorkflow : SequentialWorkflow
    {
        private IfElse ifElse1;
        private IfElseBranch ifElseBranch1;
        private IfElseBranch ifElseBranch2;

        private Code code1;
        private Code code2;

        ParameterDeclaration amount = null;
        ParameterDeclaration status = null;

        public BasicSequentialWorkflow()
        {
            InitializeComponent();
        }

        private void InitializeComponent()
        {
            amount = new ParameterDeclaration();
            status = new ParameterDeclaration();

            ActivityBind activitybind1 = new ActivityBind();
            ActivityBind activitybind2 = new ActivityBind();

            CodeCondition codecondition1 = new CodeCondition();

            this.ifElse1 = new IfElse();
            this.ifElseBranch1 = new IfElseBranch();
            this.ifElseBranch2 = new IfElseBranch();
            this.code1 = new Code();
            this.code2 = new Code();

            //
            // ifElse1
            //
            this.ifElse1.Activities.Add(this.ifElseBranch1);
            this.ifElse1.Activities.Add(this.ifElseBranch2);
            this.ifElse1.ID = "ifElse1";
            //
            // ifElseBranch1
            //
            this.ifElseBranch1.Activities.Add(this.code1);
            //
            // code condition to check request amount
            //
            codecondition1.Condition += new ConditionalExpression(CheckAmount);
        }
    }
}

```

Listing 2. Code-only WF workflow code snipped

Code-separation authoring means that the workflow model is separately defined using a declarative markup language and it is stored in a dedicated .xaml file. The actual code, for example, the code included in a code activity, is stored in a normal source code file (e.g. ending with .cs in case of C#). Interestingly, WF does not explicitly support any of the standard markup languages such as BPEL, but defines its own language called Extensible Application Markup Language (XAML). Given that fact, it is likely that the domain of process definition languages will stay fragmented and therefore the issue of interoperability addressed within this thesis remains highly important. XAML is a general purpose, XML-based markup language that describes a hierarchy of objects, in this case, activity objects within the workflow. At compile time, the markup file is compiled together with the source code file using the so-called workflow compiler. Listing 3 represents the XAML markup generated from the graphical workflow model shown in Figure 9. A large amount of markup is generated by the visual designer even for such a tiny workflow example. Obviously, such ‘noisy’ markup is not well suited to be read or written by hand.

```
<?Mapping XmlNamespace="ComponentModel" ClrNamespace="System.Workflow.ComponentModel" Assembly="System.Workflow.ComponentModel" />
<?Mapping XmlNamespace="Compiler" ClrNamespace="System.Workflow.ComponentModel.Compiler" Assembly="System.Workflow.ComponentModel" />
<?Mapping XmlNamespace="Activities" ClrNamespace="System.Workflow.Activities" Assembly="System.Workflow.Activities" />
<?Mapping XmlNamespace="RuleConditions" ClrNamespace="System.Workflow.Activities.Rules" Assembly="System.Workflow.Activities.Rules" />
<SequentialWorkflow x:Class="ExpenseWorkflows.Workflow1" x:CompileWith="Workflow1.xaml.cs" ID="Workflow1" xmlns:wcm="System.Workflow.ComponentModel" />
  <x:Attribute Name="System.Workflow.Activities.Rules.RuleConditionsAttribute" Arguments="typeof(ExpenseWorkflows.AutoApproveCondition)" />
  <EventSinkActivity ID="expenseReportSubmitted1" Invoked="ReportSubmitted_Invoked" EventName="ExpenseReportSubmitted" />
    <EventSinkActivity.ParameterBindings>
      <wcm:ParameterBinding ParameterName="Report" xmlns:wcm="ComponentModel">
        <wcm:ParameterBinding.Value>
          <wcm:ActivityBind Path="report" ID="{Workflow}" />
        </wcm:ParameterBinding.Value>
      </wcm:ParameterBinding>
    </EventSinkActivity.ParameterBindings>
  </EventSinkActivity>
  <IfElse ID="EvaluateExpenseReport">
    <IfElseBranch ID="IfAutoApproveReport">
      <IfElseBranch.Condition>
        <?Mapping XmlNamespace="System.Workflow.Activities.Rules" ClrNamespace="System.Workflow.Activities.Rules" />
        <ns0:RuleConditionReference Condition="AutoApproveCondition" xmlns:ns0="System.Workflow.Activities.Rules" />
      </IfElseBranch.Condition>
      <InvokeMethodActivity ID="approveExpenseReport1" MethodName="ApproveExpenseReport" InterfaceType="System.Workflow.Activities.IExpenseReport" />
        <InvokeMethodActivity.ParameterBindings>
          <wcm:ParameterBinding ParameterName="report" xmlns:wcm="ComponentModel">
            <wcm:ParameterBinding.Value>
              <wcm:ActivityBind Path="report" ID="{Workflow}" />
            </wcm:ParameterBinding.Value>
          </wcm:ParameterBinding>
        </InvokeMethodActivity.ParameterBindings>
      </InvokeMethodActivity>
    </IfElseBranch>
    <IfElseBranch ID="ElseRejected">
      <InvokeMethodActivity ID="rejectExpenseReport1" MethodName="RejectExpenseReport" InterfaceType="System.Workflow.Activities.IExpenseReport" />
        <InvokeMethodActivity.ParameterBindings>
          <wcm:ParameterBinding ParameterName="report" xmlns:wcm="ComponentModel">
            <wcm:ParameterBinding.Value>
              <wcm:ActivityBind Path="report" ID="{Workflow}" />
            </wcm:ParameterBinding.Value>
          </wcm:ParameterBinding>
        </InvokeMethodActivity.ParameterBindings>
      </InvokeMethodActivity>
    </IfElseBranch>
  </IfElse>
</SequentialWorkflow>
```

Listing 3. Code-separation XAML workflow markup

WF also defines the no-code workflow-authoring mode. However, this only means that XAML definitions can be modified and compiled separately as long as the required code is already available, for example, in the used base classes. If new code needs to be written, Visual Studio automatically switches to code-separation mode.

WF, as most workflow frameworks, also provides support for workflow persistence and, based on this, support for transactions. As mentioned earlier, persistence is implemented as one of the core runtime services of WF whereas registering a persistence service is optional. Users can implement their own persistence service or use the standard SQL persistence service included in WF. In any case, only one persistence service can be registered with the workflow runtime. This limits flexibility in where to store individual workflow instances. For instance, for organizational or performance reasons it can make sense not to store all workflows to one type of durable media or to store a workflow to a different media as soon as it has reached a certain state. Furthermore, WF assumes that the persistence service is the entity that decides whether a workflow should be stored persistently or not. In practice, however, it can make sense to let the workflow itself dynamically decide if it should be stored persistently or not. This way, performance and resource consumption can be optimized. For example, making sure to store workflows persistently only if absolutely necessary, and not simply for every single wait state, can increase system performance since persistence operations are typically time consuming. Finally, WF also provides means for workflow monitoring via so-called workflow tracking services. In contrast to persistence, here the user can register multiple tracking services, which all receive tracking events from the runtime engine. Events include workflow related events, activity related events, and user events. The latter can be freely defined and generated at any point in the life cycle of a workflow.

In summary, WF fulfils the basic requirement set out for this thesis to have a clear separation between the process execution engine including its generic core services and higher-level tools and languages. In other words, it is possible to use the engine without using XML definition languages and without using the comprehensive tools chain integrated into Visual Studio. As such, it could be used to implement one or more domain specific process languages on top of it to support interoperability and evolution. Unfortunately, it is tightly integrated into .NET and the Windows platform, which makes it platform dependent and thus not exactly lightweight and versatile. Nevertheless, it encourages the process-driven programming idea as it can be embedded into all kinds of Windows applications. From a programmability perspective, WF foundation is rather

traditional as it turns the workflow graph straight into an object model. As a result, constructing workflows programmatically straight via the class library is rather cumbersome as illustrated in Listing 2. This thesis project goes a step further and considers providing a workflow specific programming model rather than an object model to ease direct programming of complex process flows.

2.1.4. Java Business Process Management

In the open source domain, a large number of BPM/workflow projects exist. Most of them focus on one particular, sometimes proprietary, domain specific process language. Enhydra Shark [64] and ActiveBPEL [65] are examples for these types of projects, and there are many more. Only a few provide a generic and embeddable process engine, which does not require a particular server environment and which can be directly programmed without the use of high-level tools and code generation. JBoss Java Business Process Management (jBPM) [43] is among those and it is the most popular open source project in the BPM/workflow space. Initially, jBPM has been a typical, rather monolithic workflow engine using the jBPM Process Definition Language (jPDL) as its proprietary process markup language. Then, at the time when BPEL became more and more popular as a process definition standard, a BPEL mapping was added. Just after that, while this thesis project has already been started, the jBPM people realized that it makes sense to clearly separate the process engine core from the domain specific languages. Consequently, they came up with the Process Virtual Machine (PVM; not to be confused with the Parallel Virtual Machine using the same abbreviation) as their generic process execution kernel [37]. Figure 10 illustrates the architecture of the jBPM framework with the three domain specific process definition languages, jPDL, BPEL, and Pageflow implemented on top of PVM. Additionally, jBPM includes some optional tools such as the Graphical Process Designer (GPD) as well as task and identity management facilities.



Figure 10. The jBPM framework architecture

The process model applied in jBPM is very similar to Windows WF. The idea is to define a process graph, either using some XML markup like jPDL or visually, and then

map it to a corresponding Java object model. The PVM engine executes the graph definition by processing the corresponding object tree. Each object in the tree represents an activity node of the process graph. The actual runtime behaviour of an activity must be implemented in Java as jBPM is inherently Java based. An activity is triggered when a transition within the graph leading to the activity is taken or if an event occurs. If triggered, the activity executes relevant functional code. Furthermore, it is in full control of the further propagation of the graph execution. This means either it can trigger the next activity by taking a transition to that activity or it can enter a wait state waiting for an event to occur. Starting the execution of a process or sending an event, here called signal, to a process is always synchronous meaning that the caller's thread is used to execute activities until the process enters a wait state. If concurrency is required within a process, it is up to the activity developer to implement activities that start multiple executions of sub-trees of the graph within separate threads. The same is true for implementing activities to synchronize such parallel executions.

PVM is available as an independent class library such that processes can be programmed using the PVM API without using XML markup and the high-level tool chain. This makes the engine embeddable and versatile. However, similar as in Windows WF, building process graphs in Java code can be quite tedious. Therefore, the developers of PVM came up with the so-called *ProcessFactory*, which is sort of a domain specific language embedded in Java that is supposed to ease the construction of such process graphs. Listing 4 exemplifies how the ProcessFactory can be used to build a simple loan approval process graph. Arranging APIs this way is called a 'fluent API' [66]. The example process is comprised of four sequential activity nodes whereas depending on the *loan evaluation* activity the *wire the money* activity might be skipped or not. That is why the *loan evaluation* activity has two possible outgoing transitions. The behaviour of most nodes simply is a wait state meaning that a signal must be sent to trigger the next transition. For example, to approve the loan while the process is in the *loan evaluation* node, an external program must trigger the approve transition on the process' execution by calling *execution.signal("approve")*.

```

ProcessDefinition processDefinition = ProcessFactory.build()
    .node("accept loan request").initial().behaviour(new WaitState())
        .transition().to("loan evaluation")
    .node("loan evaluation").behaviour(new WaitState())
        .transition("approve").to("wire the money")
        .transition("reject").to("end")
    .node("wire the money").behaviour(new Display("automatic payment"))
        .transition().to("end")

```

```
.node("end").behaviour(new WaitState())  
.done();
```

Listing 4. jBPM loan approval example using ProcessFactory

The direct mapping of a process graph to a corresponding object model in addition with the support for wait states is known as Graph Oriented Programming (GOP) – a term established by jBPM. The goal is to introduce an explicit graphical program representation into otherwise standard object oriented programming languages. In addition to the graph-oriented model, PVM also supports composite activities. For example, a composite activity could include a sequence of sequential activities. This allows direct modelling of block structured languages such as BPEL or constructs such as UML super states. However, PVM does not support to model processes along the state machine paradigm as supported by Windows WF.

As many Java based process engines, PVM uses Hibernate to implement process persistence. Hibernate is an object relational mapping (ORM) framework that maps Java objects to relational databases [67]. As in jBPM each process consists of a graph of Java objects, Hibernate simply translates the complete object graph into records of a relational database. Consequently, jBPM solely relies on Hibernate and there is no clear concept of pluggable persistence providers implementing custom mappings. It only works with databases supported by Hibernate. The documentation does not provide any information on how to work with anything other than Hibernate or how to work with multiple databases concurrently.

In summary, jBPM along with the PVM represents an interesting process framework, at least for Java developers. The separation of the process engine from domain specific markup languages is in the spirit of this thesis. The same applies to the availability of PVM as a standalone Java library not depending on a J2EE server. The process model is less innovative and flexible though. It rather aims on process modelling using directed graphs rather than the more flexible and event-driven state machine modelling. It offers only limited help with concurrency and synchronization in particular when accessing shared data from parallel executions. Furthermore, PVM seems to be in a rather early state since important sections such as the ones on process variables, process history, and asynchronous continuations are still missing in the documentation. In terms of programmability, it provides a graph oriented object model similar to Windows WF and thus exhibits the same shortcomings if used without high-level tool support. Finally, the

dependency on Hibernate has negative impact on its flexibility and makes it less versatile and lightweight.

2.1.5. Yet Another Workflow Language

Within the domain of executable workflow languages and corresponding runtime systems, the majority of work is driven by the industry. For instance, most of the prominent processes languages such as BPEL, XPDL, and BPMN are industry standards [34]. Many of the efforts undertaken in the academic domain are rather targeting formal models such as Petri nets [68] or π -calculus [69] for workflow modelling and verification, for example, in [70,71,72,73]. One project, which has carried the theoretical work over into the development of a workflow runtime system is the Yet Another Workflow Language (YAWL) project originally developed at Eindhoven and Queensland University [74]. The development of YAWL was inspired by previous work on workflow patterns [75]. The idea behind YAWL is to define a workflow language that directly supports all the workflow patterns and that has a formal semantics to enable verification. Essentially, YAWL is a graphical workflow language based on Petri nets with a few extensions to support all workflow patterns. Thus, the designers claim that the language is superior compared to others in terms of expressiveness. Figure 11 shows the symbols used in the YAWL language.

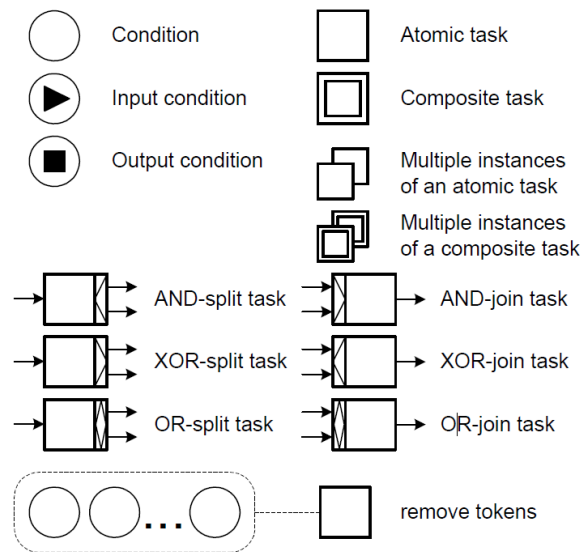


Figure 11. Symbols used in YAWL

In addition to the graphical language, a corresponding Java-based execution environment including a visual designer and an execution engine have been developed [76]. The execution engine takes yet another proprietary XML-based process language representing

YAWL diagrams as input. Obviously, the engine is not intended to be used outside the YAWL environment, for instance, embedded into some process-driven application. It requires a JSP server into which the engine is installed. Communication with the engine is carried out via a Web services interface indicating that it is designed to work in a SOA environment. For the documentation, it appears that there is no well-documented API, which supports direct programming of YAWL workflows in Java code rather than using the XML format or the visual design tools. As many Java-based workflow frameworks, YAWL requires Hibernate to support workflow persistence. An extended version called newYAWL is currently under development. It is based on the semantics of coloured Petri nets and it includes some extensions to better support different workflow perspectives such as the data and resource perspective [77].

In summary, YAWL focuses on the definition of a graphical workflow language based on Petri nets theory rather than on the design of a generic execution engine. The execution framework is rather monolithic and domain specific. Low-level programming of workflow definitions is not considered either. In fact, the YAWL language can be considered a high-level language, which could be implemented on top of the generic process framework presented in this thesis.

2.1.6. Bossa

Bossa is an open source process engine developed by a group of people from Brazil [78]. Although it is not as comprehensive and well known as jBPM, it satisfies some of the requirements of this thesis project as it claims to be lightweight and embeddable. The fact that it comes as a standard Java library not requiring any database system or server environment underpins that. As in YAWL, its process model is derived from coloured Petri nets. In Bossa, a workflow definition is called a *case type*. In case types, Petri net places are mapped to possible states of the case execution, transitions are mapped to tasks, and tokens represent the current case states. The basic idea of the Bossa engine is to strictly focus on state and control flow management as this represents the fundamental feature of a workflow engine. Additionally, it also considers the people, here called *resources*, involved in workflows. In other words, Bossa only concerns itself with the sequence of activities and with the right people to perform them. Other aspects such as data storage using a particular database system, the visualization of workflows, or higher-level modelling tools or languages are considered external and optional with respect to the core engine. This approach enables Bossa to be self-contained, embeddable, and

rather generic. Even state management is very basic. The engine only stores the case ID and the currently active task. Storing complex state data, for instance, in one ore more state variables is not supported. Work items are assigned to resources whereas resources are simply represented by IDs. Authenticating and mapping real users to these IDs is again not part of the Bossa engine but the responsibility of the application hosting it.

The Bossa engine itself does not include any XML markup for defining case types such that workflows must be defined in Java code. This means, the Petri net model must be mapped to a Java object model as exemplified in Listing 5. Alternatively, there is a separate tool, which can generate Bossa case types from Petri Net Markup Language (PNML) [79] encoded sources.

```

CaseType caseType = new CaseType("TestCaseType");

Place A = caseType.registerPlace("A", 1);
Place B = caseType.registerPlace("B");
Place C = caseType.registerPlace("C");
Place D = caseType.registerPlace("D");
Place E = caseType.registerPlace("E");
Place F = caseType.registerPlace("F");
Place G = caseType.registerPlace("G");
Place H = caseType.registerPlace("H");

Transition a = caseType.registerTransition("a", "requesters");
Transition b = caseType.registerTransition("b", "sales-$a");
Transition c = caseType.registerTransition("c", "directors");
Transition d = caseType.registerTransition("d", "sales");
Transition e = caseType.registerTransition("e", "sales");
Transition f = caseType.registerTransition("f", "$a");

a.input(A, "1");
a.output(B, "1");
b.input(B, "1");
b.output(C, "!SOK");
b.output(D, "SOK && DIR");
b.output(E, "SOK && !DIR");
c.input(D, "1");
c.output(B, "ADIR == 'BACK'");
c.output(E, "ADIR == 'OK'");
c.output(H, "ADIR == 'CANCEL'");
d.input(E, "1");
d.output(F, "1");
e.input(F, "1");
e.output(G, "1");
f.input(C, "1");
f.output(B, "OK");
f.output(H, "!OK");

HashMap attributes = new HashMap();
attributes.put("SOK", new Boolean(false));
attributes.put("DIR", new Boolean(false));
attributes.put("ADIR", "");
attributes.put("OK", new Boolean(false));

```

```
caseType.buildTemplate(attributes);
```

Listing 5. Creating a Bossa case type

As already shown with Windows WF and jBPM, defining workflows by creating object trees is rather tedious. Firstly, all the places including the initial token placement as well as the transition objects must be created. As transitions represent tasks, the group of resources that should execute these tasks can be specified when defining transitions. Secondly, the input and output of the transitions must be connected to the places. At this point, Bossa allows defining the edge weight as a JavaScript integer or boolean expression. Finally, the initial values of the case variables must be defined to build the final case type template.

During case execution, the host application simply asks the Bossa engine for resources that have pending work items. Furthermore, the host application must tell the Bossa engine when work items are opened and finished. If a work item is finished, the Bossa engine will reflect that by moving the tokens of the effected case, this will eventually activate other work items and the case will progress. Thread management for executing work items concurrently is still under the responsibility of the host application.

Bossa includes transparent persistence support. This means, if persistence is switched on, the state of the process engine and its cases is saved to a given directory in the file system automatically. For workflow monitoring, Bossa provides a notification bus facility where listeners can be registered to receive relevant workflow or resource events.

In summary, Bossa is very interesting as a lightweight workflow kernel. Therefore, this thesis shares a few ideas with Bossa, for instance, the distinct focus on state and control flow management. However, the Petri net-based process model, the notion of resources, and the transparent persistence support limits flexibility and versatility of the engine.

If the sequence of tasks is not precisely known at design time, a state machine based process model is clearly superior. In addition, persistence cannot be configured on fine granularity and the concept of multiple pluggable persistence providers is not supported either. Finally, the classical approach of mapping the Petri net graph to a Java object model annotated with some JavaScript conditions does not ease programmability.

2.1.7. OpenSymphony Workflow

OpenSymphony Workflow (OSWorkflow) is another process engine coming from the open source community [80]. It is Java based and requires a Java servlet container or full J2EE server to run. Obviously, this makes it only embeddable into server-based

applications. Nevertheless, it is interesting as its process model rests upon the theory of finite state machines. OSWorkflow provides its own proprietary XML markup format for defining workflows in terms of state machines. In this connection, states of a state machine are called *steps* and transition functions are called *actions*. Listing 6 shows a code snippet from a workflow definition using OSWorkflow markup. The workflow consists of two steps whereas the first step has two actions. The first action does not change the step but only updates a string representing the workflow status. The second action additionally progresses the workflow to step two. Actions can be decorated with conditions and functions that are invoked if the conditions are met. Conditions as well as functions can be implemented in Java classes implementing a specific interface or via BeanShell [81] scripts. In the latter case, the script code is embedded directly in the XML markup.

```

...
<step id="1" name="First Draft">
  <actions>
    <action id="1" name="Start First Draft">
      <restrict-to>
        <conditions>
          <condition type="class">
            <arg name="class.name">
              com.opensymphony.workflow.util.StatusCondition
            </arg>
            <arg name="status">Queued</arg>
          </condition>
        </conditions>
      </restrict-to>
      <pre-functions>
        <function type="class">
          <arg name="class.name">
            com.opensymphony.workflow.util Caller
          </arg>
        </function>
      </pre-functions>
      <results>
        <unconditional-result old-status="Finished" status="Underway"
          step="1"/>
      </results>
    </action>
    <action id="2" name="Finish First Draft">
      <results>
        <unconditional-result old-status="Finished" status="Queued"
          step="2"/>
      </results>
    </action>
  </actions>
</step>
<step id="2" name="finished" />
...

```

Listing 6. OSWorkflow XML workflow markup snippet

Thread management is not part of OSWorkflow. Users must query the engine for available actions and provide threads to trigger them. Furthermore, OSWorkflow provides a pluggable persistence-store provider mechanism and it includes a number of standard implementations, for instance, an in memory store, a Java Database Connectivity (JDBC) store, or an Enterprise JavaBeans (EJB) store. Custom implementations are possible as well, however, only one store can be registered with the engine at any given time. The data that is stored persistently in OSWorkflow when persistence support is turned on includes basic information about the existing workflow instances, the previous and current workflow steps, and the so-called *PropertySet*. The latter can be used to store arbitrary instance specific data persistently. OSWorkflow does not provide a dedicated monitoring interface; developers must build it themselves based on the persistent store.

In summary, OSWorkflow is a quite generic process engine, which also is embeddable. However, it poses a number of requirements to the host application. For example, it relies on multiple other open source software packages and it requires a Java servlet container to run. Furthermore, the XML based workflow definition language possibly mixed with BeanShell scripts is not only proprietary but also quite inconvenient to edit manually. These factors clearly limit the versatility and programmability of the framework.

2.2. AUTHENTICATION AND TRANSACTION AUTHORIZATION METHODS

2.2.1. Introduction

While relevant attack scenarios are presented in Section 1.3 (c.f. Figure 4), this section firstly provides an overview of state-of-the-art remote authentication schemes and the available client security devices used by them. Secondly, a threat-solution taxonomy relating the schemes to the attack scenarios is developed. Finally, a number of solutions that potentially provide protection against MSW attacks and are thus particular relevant for this thesis are presented and discussed with respect to the challenges and requirements set out in Section 1.8.2.

2.2.2. Remote Authentication Schemes

Any remote authentication method's goal is to establish and secure an authenticated information channel by proving a user's identity through an associated security channel. For most methods, the information channel also serves as the security channel and, unless stated otherwise, it is assumed in the following discussion. It is also assumed that it is always the client who connects to and authenticates with a server.

Once an authenticated information channel is established, it might be required to further secure individual transactions sent over this channel via transaction authorization. Similar to authentication where the information channel is authenticated, here, transaction details must be authenticated. Thus, the method used to implement transaction authorization can mostly be derived from the authentication method used. For example, instead of digitally signing an information channel specific challenge for authentication purposes, transaction specific information can be signed, for example, using public key cryptography. If transactions are authenticated via digital signatures, the system is considered an appropriate means for non-repudiation and is oftentimes called transaction signing. This is not the case if shared secrets such as passwords or secret keys are used.

2.2.2.1. Static Passwords

The oldest, most primitive remote authentication method is the use of static passwords, which typically change, at most, only every few months. With this method, a client presents a single static password to the server for each authentication; the server then matches it with the password stored for that client. Static passwords are still widely used in application domains where the environment is well controlled, the protected values are limited, or the potential risks are manageable. Example domains include authenticating a user locally with his or her personal computer (PC); remote authentication within local-area networks or intranets; or access control to an Internet bookstore. However, when it comes to highly sensitive data such as information about financial institutions, their customers, and their transactions, researchers deem static passwords an insufficient remote authentication method [23].

2.2.2.2. One-time Codes

Remote authentication with one-time codes is based on the idea that both client and server share a secret. The client presents it to the server either as is (that is, the secret is the one-time code), or in a derived form according to some algorithm, possibly with additional data also known to the server. An exception here is with systems based on one-way-hash functions such as the S/KEY system [82]. In these systems, rather than using a shared secret, the server authenticates the next code in a sequence based on the previous code. In the one-time code approach, clients present each code to the server only once; codes cannot be reused.

Scratch lists. A scratch list is the simplest form of a one-time code. A scratch list is typically given to the client once, in paper form, and usually contains about 40 to 100 codes. The server knows these codes, and clients use them sequentially or in an indexed form. So, the shared secret is the listed code and clients use it as is, without further derivation. If the client uses an indexed scratch list, the server decides which one-time code should be used next by specifying its index in the list. Otherwise, clients typically have to track the used codes themselves. Either way, each code is used once and only once, and the server automatically sends the client a new list when only a certain number of codes are left. Figure 12a illustrates a scratch list scenario.

Short-time codes. With short-time codes, both client and server share one or more secrets exchanged in advance. They might, for example, use a symmetric key, and derive one-time codes for authentication based on these shared secrets and the current time. They never actually exchange the shared secrets, just the codes derived via some derivation function $f(x)$ (see Figure 12b). Time granularity is typically on the order of a few minutes, that is, the same code is derived during that time. This permits small time shifts between the client and server, and the server also usually accepts codes derived from times within the previous and next time slots.

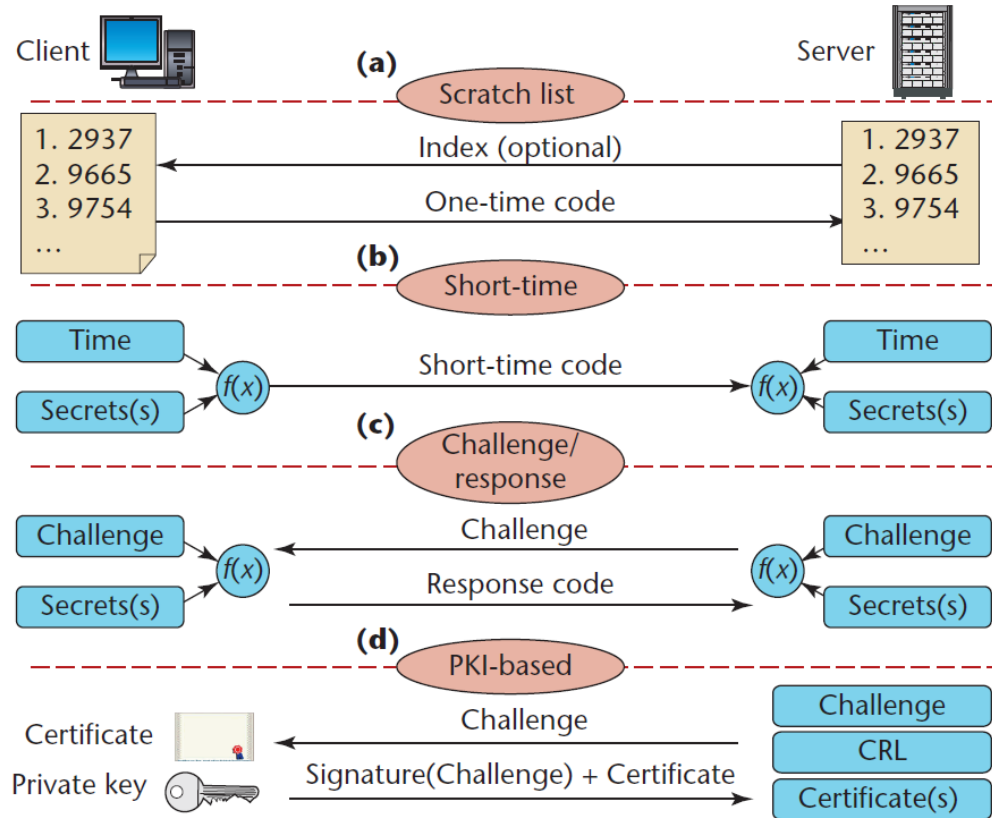


Figure 12. The four primary authentication schemes

Challenge/response codes. As Figure 12c shows, challenge/response codes modify the short-time codes concept by substituting a server-specified challenge for the current time. That is, client and server are again initially equipped with one or more shared secrets, such as a symmetric key and a counter value that is incremented after each authentication attempt. Then, for authentication, the server presents a randomly chosen challenge to the client. The client then responds with a code derived from the shared secrets and the challenge, while the server performs the same derivation; the counter value thereby prevents identical responses to the same challenge. Although there is no time-shift problem with this approach, the client can still inadvertently calculate response codes, potentially misaligning some secrets shared with the server. Equally, the server might be accidentally or intentionally triggered to send out challenges and calculate response codes, which again misaligns shared client-server secrets. Given this, the server typically does not accept one and only one response code; it also accepts codes neighbouring the target code up to a certain limit (such as the five previous and following ones). If it detects a match, the server realigns its shared secrets accordingly.

Alternatively, the server might send the challenge via a separate security channel that is authenticated by some other means (such as an SMS communication secured via the mobile phone network). Thus, the identity function can derive the response and no shared secrets are required. In this case, the resulting authentication's strength is inherited from the selected security channel.

2.2.2.3. *Public Key based Authentication*

In contrast to one-time codes, authentication based on a so-called public-key infrastructure (PKI) utilizing public-key cryptography does not rely on shared secrets [83]. Instead, each client is initially equipped with a private key (never to be exposed) and a matching public key. Furthermore, the server uses a PKI that issues a digital certificate to bind the client's identity to his or her public key. The certificate contains the client's public key, which is signed by one or more certificate agency (CA) that the server trusts. Although it is somewhat difficult to establish and maintain a PKI, the authentication itself is rather simple. The server presents a randomly chosen challenge, and the client signs with its private key. (If both parties fail to use necessary safeguards to prevent well-known crypto-analytic attacks, such as the chosen-plaintext attack, however, then the authentication scheme can be broken.) As Figure 12d shows, both the signed challenge and the client's certificate are then returned to the server in response. The server thereupon ensures that:

- the client's certificate is valid (that is, it is signed by a trusted CA and the signature verifies), and
- the signature of the challenge verifies with the given client certificate.

The server also maintains a list of revoked client certificates, the so-called certificate revocation list (CRL), in case, for example, a client's private key is compromised and must be invalidated. During authentication, the server checks each client-presented certificate against the CRL and if it finds a match, it denies the authentication. For a positive test, or if more than one server uses the same CA without necessarily authenticating the same group of clients, the server usually stores either copies of the certificates or corresponding hash values that it can authenticate. Furthermore, each certificate's lifetime is usually limited from a couple of months to a couple of years, after which the server issues a replacement certificate to the client.

2.2.2.4. *Biometrics*

Biometrics base remote authentication on “being something” instead of “knowing something” (such as a one-time code) or “having something” (such as a private key). In biometrics, the server matches one or more client biometrics such as a fingerprint, facial feature, iris pattern, or hand geometry with the same information previously stored at the server during enrolment. During the enrolment process, the serving organization captures and stores each client’s biometric information under well-controlled conditions. Then, for authentication, the client again captures that information and sends it and the claimed identity to the server for matching. The server can thus reliably authenticate clients given three assumptions. That is, that biometric information

- can be reproducibly captured repeatedly,
- cannot be easily faked, and
- is sufficiently different between any two clients.

Unfortunately, clients cannot capture biometric information reproducibly, but rather only in close approximation. A biometric match never returns a clear-cut yes or no result, it returns only a probability as to the verifiability of the client’s identity. This coercively causes so-called “false rejects” (genuine clients are not authenticated) and “false accepts” (impostors are authenticated). Although developers can move a threshold to adjust the tendency as to which side a method will fail and with what probability, each biometric authentication is inherently prone to this type of misjudgement.

A significant drawback here is the possibility to obtain some biometric properties after physical contact (fingerprint on a water glass, for example) or by using a high-resolution picture of a person’s eyes. This limits biometrics’ value in scenarios where forgery of this type, say, plastic fingers or photographs presented during the authentication phase, are undetectable. Also, like static passwords, clients can use biometric features repeatedly once they are obtained. To prevent such misuse, one would have to authenticate the biometric device used to capture the biometrics to ensure the authentication data’s origin [84]. This would introduce a completely new complexity level, making biometrics of limited value for remote authentication over insecure networks.

2.2.3. **Client Security Devices**

The most common client hardware is a standard desktop PC, which is also an easy platform to attack. People thus often additionally use a security device, such as a smart card, either in a standalone reader or one that connects to the PC, a mobile phone or

PDA, or a smart memory stick. The security device's software then, at least partially, performs the authentication, preventing certain types of attacks against the PC. As described in the following, there are currently several commonly used client-security devices. Although this might make the security channel distinct from the information channel, we assume that the information channel is always established between a client's PC and the server.

2.2.3.1. Smart Cards

A smart card consists of a plastic card with a small embedded microprocessor with various memory types such as ROM, RAM, or EEPROM and tamper-resistant properties such as secure crypto-coprocessors for symmetric and public-key cryptography. Typically, smart cards have external power and clock, and communication is serial, either contact-based or contactless.

To communicate with the smart card, users need a reader that connects it with either a PC or stand-alone device. Stand-alone readers typically have at least a small display and a numeric keypad where users enter their personal identification number (PIN) and commands. Readers providing a PC connection are commonly classified according to their capabilities, with class 1 readers simply providing connection, class 2 adding a pinpad, and class 3 readers offering a display and some programming capabilities. Class 4 readers feature a separate security module and a virtual machine for custom application execution; thus, they are considered secure execution platforms. Given their resistance to tampering, smart cards are generally accepted as sufficiently secure to store sensitive data, particularly private keys. Ideally, private keys are generated on the smart card and are never directly exposed to the outside world.

2.2.3.2. Mobile Phones and Personal Digital Assistants

Mobile phones or personal digital assistants (PDA) are separate computing platforms with their own displays and keypads. Both can serve either as stand-alone devices, for storing and computing one-time codes, for example, or as a PC connection using Bluetooth, infrared, or USB to remotely authenticate users. Mobile phones also can use their mobile networks as security channels.

Functionality-wise, mobile phones are increasingly general-purpose computing platforms that support more-or-less open software execution platforms. By far the most prevalent code execution platform is Sun's Java, which is used in millions of phones. Downloading

and installing Java applications is simple; vendors have a huge commercial interest in making any software purchase (such as games) easy for mobile phone users. All versions of J2ME - the stripped-down “embedded” Java version run in mobile phones – lack desktop and enterprise Java security features such as the byte-code verifier, which performs static code and integrity checks [85]. Without this on-device verifier, attackers can write “subversive” code and thereby access the data of other Java code such as a banking application residing on the same mobile phone. Only the most recent J2ME version (MIDP 2.0) includes the possibility of digitally signing code and thus tying code executed on the device to a trusted source that presumably performs proper off-device byte-code verification. Only devices that conform to this specification level meet the recommendation to only run code of known origin. It is highly dubious, however, that it is possible to educate typical users enough to verify that a piece of code’s digital signature refers to a trusted source. The underlying issue of checking certificate origin, that is, checking SSL certificates to avoid MITM attacks, has already proven intractable in PC-based online banking.

Java’s prevalence, combined with a nonexistent or difficult-to-verify code-origin check and a weak code security model, makes mobile phones far weaker than PCs when it comes to security. Storing secret information such as private keys to support remote authentication on a mobile phone that lacks a security module, that is, one that uses *soft tokens*, or software-only authentication implementations, is far too dangerous and users should not even consider it. Instead, they should use a smart card as a tamper-resistant security module. Nearly all mobile phones have a security interface module (SIM), which are smart cards that, among other things, authenticate the mobile phone with the mobile network provider. Network operators can deploy additional applications on these SIMs, which can use the mobile phone network as a security channel. The new generation of mobile phones also have a second smart card that offers secure storage for applications running on the phone’s run-time platform. The card also provides a run-time platform for secure applications. These new phones can connect smart cards to PCs using near-field communication (NFC) [86], a short-range contactless communication interface and protocol based on ISO proximity card standards [87]. Although it is convenient for users, such direct communication between NFC-enabled mobile phones and PCs requires them to add a contactless reader device to their PCs. To use the second smart card simply, that is, as a phone-based authentication with user-interaction on the phone, is similar to using the SIM. The only difference is the chip’s availability, which is rare compared to SIM,

and accessibility, which is better than SIM. Still, short of the phone-internal chip that drives all external user interactions, through a SIM Application Toolkit (SAT) [88] application, for example, using a security device helps protect only the secret data. It can not guarantee overall application integrity. If a user can be tricked into providing the password to the secret data, a malicious on-phone application can freely access the data, even if it is safely stored. Any application that uses a mobile phone as a security device must account for this fact when designing what on-phone code can do with the secret information once it is unlocked. As an obvious example, it should be impossible for an application to copy out this data, even if the user presents the correct PIN code.

2.2.3.3. *Smart Memory Sticks*

Memory sticks equipped with a smart card, in, for example, a USB form factor for use with PCs, are a rather new development. The memory stick mounts as a write-protected volume (like a CD) and usually contains some immutable software for remote authentication. This software might be a Web browser, such as Firefox, that is restricted to connecting only to certain Web sites. For added convenience, users can configure some or all of this software to automatically launch whenever the memory stick is mounted. Any mutable state and any data that cannot be exposed such as a shared secret or a private key are stored and processed on the embedded smart card. One of this technology's main challenges is how to create user-friendly updating of the read-only memory if, for example, security patches require installation of a new version of the memory stick's software. Furthermore, not all applications can operate from a read-only device. As a result, they must be temporarily copied to the PC's hard drive prior to each execution. In principle, smart memory sticks can work with mobile phones, by inserting them into the secure digital (SD) card slot, for example, at the time of writing, however, no such product could be found.

2.2.4. **Threat-Solution Taxonomy**

The threat-solution taxonomy shown in Figure 13 puts the most popular authentication schemes into relation to the security level they provide with respect to phishing, MITM, and MSW attacks. Static passwords (PW, also called code), soft tokens such as PKI certificates stored on the hard disk, as well as scratch lists are inherently prone to phishing attacks. Attackers can obtain and use them at any time, not only while the genuine users are using them. Thus, these methods do not cross the so-called *offline credential stealing attack boundary* as indicated in Figure 13. Scratch lists, even indexed

ones, do not withstand phishing attacks because the one-time code's validity period is itself rather long (or even unlimited). Also, the code is not specifically connected with a particular information channel. So, when attackers get a one-time code from a scratch list, they can use it with any information channel to the genuine server any time after an attack and prior to the legitimate user's next attempt to authenticate with that server. Biometric authentication, which is seldom used and thus not considered in the taxonomy, is also conceptually vulnerable to phishing attacks for the same reasons. In contrast, timer-based short-time codes at least limit an attacker's window of opportunity to a couple of minutes making phishing attacks largely impossible. Still, only challenge/response authentication effectively prevents phishing by strictly associating each response to a specific authentication attempt. Applying this line of reasoning, PKI-based authentication methods also prevent phishing attacks. In fact, we can consider the server challenge's digital signature as a response, much like a one-time code challenge/response scheme, though the latter usually employ a simpler infrastructure than PKI.

Although effective against phishing, short-time PW are not resistant against MITM attacks and thus are not crossing the so-called *single online channel breaking attack boundary* (c.f. Figure 13). Single online channel means that the information channel is also the security channel. To cross that boundary SSL/TLS client authentication using a security device such as a class 1 smart card reader and a corresponding smart card is required. Keeping private keys on the smart card thwarts phishing, while the SSL/TLS client authentication renders MITM attacks impossible. Unfortunately, the SSL/TLS protocol does not support one-time code schemes for client authentication, although researchers have made a proposal in that direction [89]. On the application level, MITM attacks can be prevented only by challenge/response one-time codes or PKI-based authentication methods - if both are extended to this end. To exclude a MITM, the client and server must uniquely identify the information channel, and then use this identification as an additional input parameter when calculating the one-time code response, or concatenate it with the data to be digitally signed. The client and server could, for example, use the session-specific SSL/TLS protocol information – such as the handshake message's hash value - to identify the information channel. Such information would be different for client and server if, instead of one end-to-end session, they had two sessions with an MITM. Consequently, either the client's one-time code response

or the signed data wouldn't match, respectively. If the session identification is independent of the SSL/TLS connection, as in the use of cookies, for example, the session has no MITM resistance [90].

Although TLS client authentication in combination with a hardware token, for example, a public key (PK) enabled smart card, protects against MITM attacks, it does not protect against MSW attacks. A virus or Trojan horse located on the client machine to which the smart card is connected can misuse the card. For instance, it could use the card for authentication or transaction signing transparently for the user. One approach to prevent such MSW attacks is to authorize/sign transactions via a second online channel such as the short message service (SMS) to the user's mobile phone. In this multi-channel scenario, the Internet connection between the PC and the server is used as the information channel and SMS communication is used as the security channel. This approach is considered resistant against MSW attacks under the assumption that the attacker is not able to gain control over the security channel as well. However, this assumption does not necessarily hold, since a normal mobile phone is not considered a security device. Therefore, this method does not cross the so-called *multi online channel breaking attack boundary* as indicated in Figure 13. To achieve this, transactions must be signed on a trusted hardware device, for instance, a class 3 or 4 smart card reader including a secure display and pin pad.

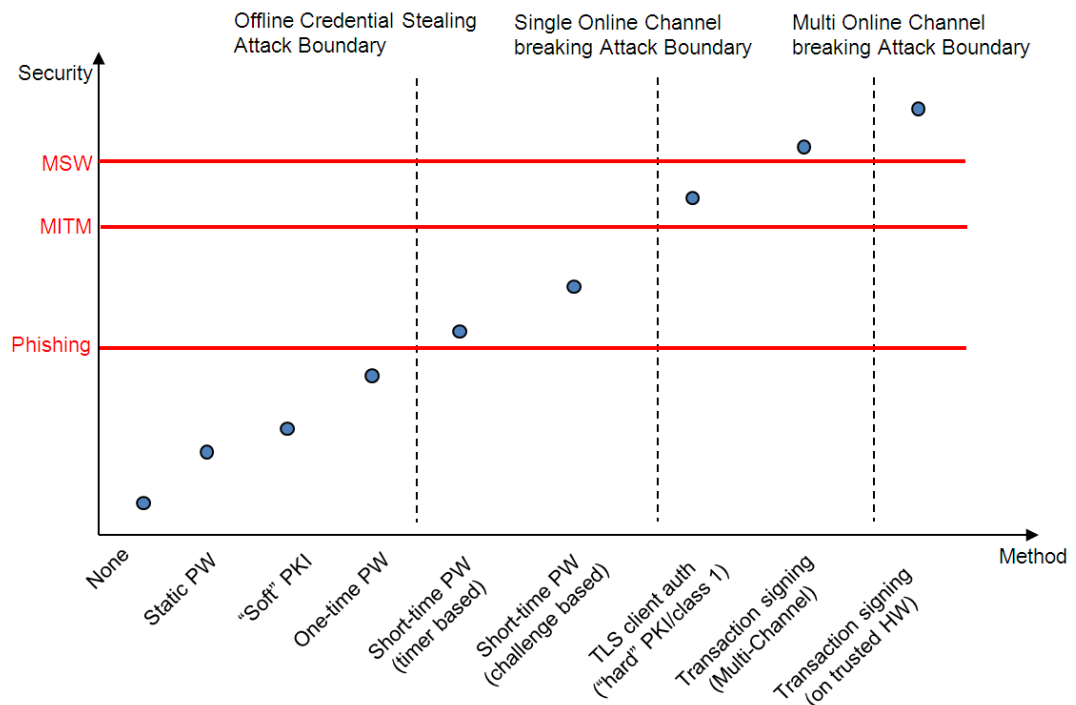


Figure 13. Authentication thread-solution taxonomy

The majority of solutions, which are in use at the time of writing, can be classified around the junction of the phishing line and the offline credential stealing attack boundary line shown in Figure 13. Only a few are resistant against MITM and even less provide transaction signing on trusted hardware such that all three attack classes are rendered impossible. Unfortunately, recent research shows that while online services are becoming more resistant to phishing attacks - such as by moving from scratch lists to short-time password-generating hardware tokens - MSW attacks are increasing [25].

In the following sub-sections, those methods that claim to be resistant against phishing, MITM, and MSW attacks are discussed in more detail. Furthermore, they are related to the challenges and requirements set out in Section 1.8.2, which include more aspects than sole security.

2.2.5. Mobile Transaction Authentication Number

The mobile transaction authentication number (mTAN) system is based on the multi-channel approach. It uses SMS communication to a user's mobile phone as the security channel. Figure 14 illustrates the mTAN usage scenario. As usual, the user connects to the remote server via Internet, typically, using a standard Web browser. Then, during user authentication, a server-generated one-time password (the TAN) is sent to the user's mobile phone via SMS. To complete authentication, the user must manually copy the TAN from the phone's display to the Web browser and submit it to the server. Similarly, when the user has submitted some sensitive transaction, a transaction-specific server-generated TAN is sent to his or her phone along with the details of a given transaction (c.f. Figure 14). The user can then verify the transaction details and approve it by copying the TAN to the web browser. The mTAN system considers the mobile phone as a trusted platform with respect to the Internet service since it is unlikely that an attacker has gained control over both, the client PC and the mobile phone. Under this assumption, the system is resistant against MITM/MSW attacks as long as the user carefully examines the transaction details. However, today's mobile phones increasingly become open multi-application platforms connected to the Internet and, as such, will eventually face similar attack scenarios as PCs. Additionally, the mTAN system introduces a substantial privacy and confidentiality issue since short messages sent to the phone can be traced, at least by the mobile network operator(s) involved. Also, most users do not enable PIN protection to restrict physical access to their mobile phones while they are switched on.

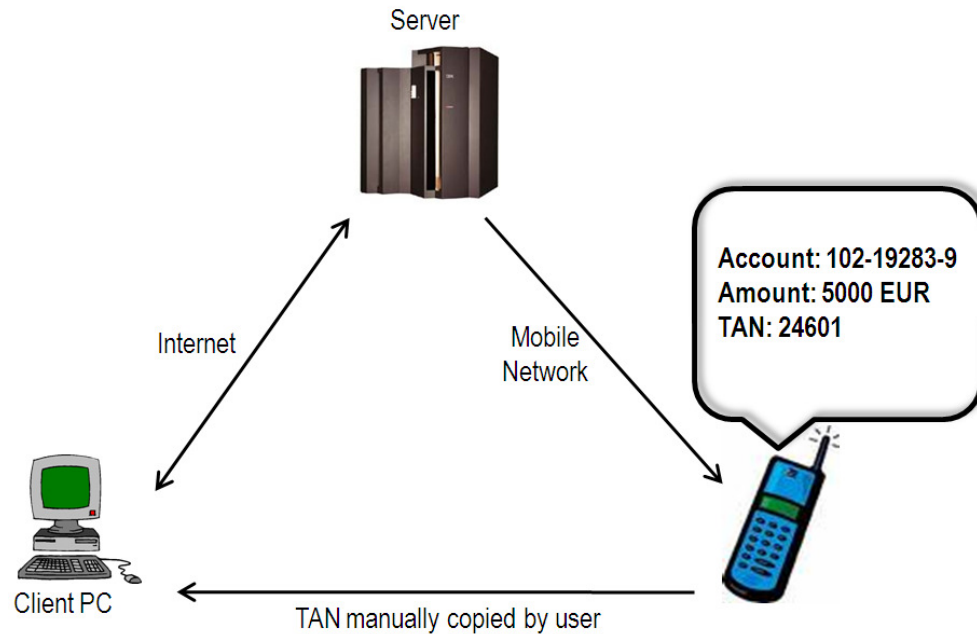


Figure 14. mTAN usage scenario

With respect to ease-of-use and mobility, the mTAN approach also exhibits some shortcomings. Manually copying a password not only for authentication but for each transaction is rather inconvenient. Mobility seems to be guaranteed as the approach works independent of the PC used, for instance, when connecting from an Internet café. However, the mobile network connection might be limited depending on the location, for instance, in buildings. Also, SMS delivery is by definition best effort and not reliable. Finally yet importantly, the SMS communication generates additional costs per authentication/transaction, which might sum up to a significant amount considering a large and active user base.

With respect to ease-of-integration and administration, obviously, the system requires some server changes in order to generate and distribute the short messages. However, the fact that service providers do not need to distribute devices, clearly makes the system relatively easy and cost efficient to introduce. On the other hand, the multitude of different mobile phones generates a lot of support costs as users, for instance, will not be able to operate them properly and text messages will look different on the different phone models. Overall, it is a system, which can be introduced quickly and with limited costs, but it lacks security, privacy, user convenience, and maintenance cost in the long term.

2.2.6. Standalone Challenge/Response Reader

Standalone challenge/response devices that include a smart card reader are widespread, at least in Europe, for a number of years already. However, in most cases they are used as standard challenge-based short-time password generators. As such, they are only suitable to protect against phishing attacks but not against MITM/MSW as indicated in Figure 13. To extend them towards this end, it is not only used for authentication, where the user must enter a server-generated challenge and the device calculates a corresponding response code with the help of a smart card, but also for transaction authorization. In the latter case, the user must, in addition to the random challenge, also enter critical transaction details such that the resulting response code becomes dependent on them. Consequently, MSW can no longer silently modify transactions sent to the server, as the server would detect this when verifying the response code.

In order to make the reader device itself user independent, the cryptographic keys required to calculate response codes are usually stored on a smart card. Users must insert the smart card and enter the card PIN before using the device for authentication or transaction authorisation purposes. Furthermore, in order to re-use existing infrastructure, in many cases already existing cards are used. This is possible because most of the credit and debit cards support the card authentication program (CAP) standard specified by Europay, MasterCard, and Visa (EMV). Therefore, such challenge/response readers are oftentimes called *EMV CAP reader*. As an example, Figure 15 shows the reader distributed by the Barclays Bank to their online banking customers.



Figure 15. Barclays EMV CAP reader

Concerning the effectiveness against software attacks as discussed in 1.8.2.1, challenge/response reader-based solutions are good if the user manually enters transaction details. Since the devices are not connected to the PC, they are by definition resistant against MSW attacks targeting the device itself. In addition, it is not possible to install any software on them, which could potentially be malicious.

Mobility and ease-of-use is rather limited. In principle, the approach is perfectly mobile since the device is independent of the PC and does not require any network connection, as for instance, the mTAN solution. Unfortunately, the reader devices are usually quite bulky because a pin pad is required. Consequently, users do not like to carry them around all the time if used only occasionally. Ease-of-use represents the major issue, since the approach requires a significant amount of manual work. The challenge must be manually copied from the PC to the device, transaction details must be entered, and the response must be copied back to the PC. In essence, the solution requires too much effort for frequent users and is too complicated for occasional users.

Integration obviously requires a few changes on the server side to generate challenge values and verify response codes. The client PC does not require any software installation. Only the reader devices and smart cards, if not already deployed, must be issued. Reader devices are reasonably cheap with a price around five GBP. A weak point is the fact that the reader device software cannot be remotely updated. Overall, the approach solves the core security challenge, but does hardly fulfil any of the other requirements, in particular, ease-of-use.

2.2.7. Flicker Devices

Standalone devices have the advantage that they are independent of the client PC. This eliminates many sources of error such as connection or software issues, for instance, due to missing physical ports, operating system support, or device driver problems. On the other hand, manual interaction thwarts user convenience as described in Section 2.2.6. An innovative alternative is the communication via flickering images, originally invented at the Berne University of Applied Science in Biel, Switzerland. The basic idea is to separate the information channel and the security channel as done with the mTAN solution. However, instead of using a separate physical communication network, the security channel is tunnelled through the information channel via a flickering image representing encrypted information. In other words, in case of authentication or transaction authorization, a server-generated flickering image is embedded into the web

page and thus visible at the client PC. The image works like a bar code but has an additional dynamic component such that more data can be embedded. Essentially, it consists of black-and-white flickering vertical bars. When the image appears, the user must hold his or her device in front of the computer screen just over the flickering bars. Figure 16 shows this for the so called AXS-Card device from Axsionics [26]. The device reads the data from the image via some light sensitive diodes located at the backside of the device, decrypts it and displays it on a small embedded display. This way, the challenge as well as transaction details can be securely transferred to the device such that the user does not need to enter them manually. The device then generates a corresponding short-time password. Since the communication via the flickering image is unidirectional, users are still required to manually copy the password from the device display to the PC. In case of the AXS-Card, the device is additionally equipped with a fingerprint reader such that no pin pad is required for entering the device PIN. At the time of writing, the first EMV CAP reader devices (c.f. Section 2.2.6) extended with support for flickering communication have just appeared.



Figure 16. Flickering communication with AXS-Card

The properties of this approach are similar to the properties of the challenge/response reader solution discussed above, except that less manual input is required such that ease-of-use is clearly enhanced. However, the device must be held in front of the flickering image for more than ten seconds since the bandwidth of the image is limited. Additionally, it must be positioned such that the diodes are located over the bars and this position must be held steadily. In practice, this creates problems with interrupted data transmissions since some people do not have the fine motor skills to place and hold the device correctly. Essentially, the manual data input via a pin pad is traded against some

balancing act. Furthermore, the flickering image is not very pleasant to look at and might even cause problems with people suffering from epileptic seizures. Overall, the solution can be considered as a variation of the challenge/response reader solution where ease-of-use is still limited.

2.2.8. Secoder and Financial Transaction IC Card Reader

The classical approach to thwart MITM and MSW attacks is the use of a class 3/4 smart card reader device in combination with a smart card supporting digital signature generation. The Secoder [28] standard defined by the German institution Zentraler Kreditausschuss and the FINREAD (FINancial Transaction IC Card READER) [46] standard created as part of an EU research project are examples for such systems. In contrast to the solutions discussed previously, the card reader device is connected with the client PC, for example, via an USB cable (c.f. Figure 17).



Figure 17. FINREAD compliant class 4 smart card reader

The main idea behind the approach is that sensitive operations are sent from the PC to the reader device for verification and digital signing before they are forwarded to the server. This requires some software components to be installed on the client PC. In most cases, a specific device driver and some software components that enable communication between the Web browser and the reader device, for example, a browser extension or plug-in, must be installed. Authentication and transaction requests are then sent to the reader, the PIN for unlocking the smart card must be entered on the reader, the reader explicitly requests the user to approve operations via the embedded pin pad, and finally requests are signed by the smart card. To prevent MSW attacks, which can potentially be launched against the reader device from the client PC, the reader does not allow direct usage of the smart card. Only higher-level commands such as requesting a signature of some given data can be sent from the PC to the reader. This way, the reader can ensure that data is securely displayed and approved by the user via the embedded display and

pin pad before it is signed. No MSW running on the PC can disrupt this sequence. In addition, the smart card PIN is never exposed to the PC.

With respect to effectiveness against software attacks, class 3/4 smart card reader solutions are very good as long as the reader software is resilient against attacks from the PC. For example, malicious PC software should not be able to force the reader into malfunctioning by sending corrupted commands over the USB link.

Besides strong security, the approach also offers a very high level of convenience. Users only need to enter the card PIN on the reader and press one button to approve or decline transactions. The main reasons why these type of solutions have not become widespread lie in the cost, mobility, and maintenance domain. Firstly, with a price tag around 30 GBP, the devices are too expensive to be distributed to users free of charge. Secondly, the fact that software must be installed on the client PC creates many issues in terms of customer support, ability to use the solution with various different operating systems and Web browser platforms, as well as mobility. The latter is further decreased due to the rather bulky devices. Most readers are built for desktop usage and thus are not suitable to carry them around.

2.2.9. Mobile Identity and Sentinel

Recently, a few new solutions based on a so-called *hardened browser* have appeared. The idea is to provide a dedicated Web browser that only works for specific applications of specific service providers, for example, a browser used for online banking with a specific bank only. Such a restriction allows stripping down browser functionality to an absolute minimum and thus minimizing possibilities for attackers. For instance, such a browser must only connect to pre-configured URLs, accept only specific SSL/TLS certificates, must not support browser extensions, and so on. Additionally, scrambling and encryption tools are applied to the browser code such that runtime attacks become more complicated.

To further secure the hardened browser against manipulation, it is stored on a write protected USB stick hosting some flash memory. This enables mobility and plug-and-play convenience. Examples for such solutions are *CLX.Sentinel* [91] from Crealogix and *Mobile Identity* (mIDentity) [92] from Kobil. The latter additionally includes a smart card to securely store cryptographic keys as indicated in Figure 18.



Figure 18. Kobil mIDentity USB stick

Manufacturers claim that these solutions thwart MSW attacks. However, since the hardened browser is still executed on the client PC, this is highly questionable. Considering today's standard PC hardware and operating systems, it is not possible to extensively protect software running in such an open environment from MSW. Even if the keys used for SSL/TLS client authentication are stored on a smart card, as with mIDentity, the SSL/TLS connection still terminates on the PC and thus is subject to attacks. Furthermore, there is no secure display and pin pad. Effectively, such solutions provide, at best, a similar security level as a standard class 1 smart card reader solution plus a more difficult to attack, but still vulnerable Web browser.

Additionally, the need to transfer a full web browser from the USB stick to the PC for each online session may well introduce user acceptance problems. Promptly updating the web browser on the USB stick to keep up with released security fixes to protect against known vulnerabilities of the browser remains an administrative challenge. Overall, the advantages in the area of ease-of-use and mobility are rendered useless by the fact that the effectiveness against MSW attacks is questionable.

2.3. SUMMARY AND DISCUSSION

From the previous discussion of the relevant process execution frameworks, it is clear that none manages to fully meet the requirements presented in Section 1.8.1. The large BPM product suites from big industrial players, as discussed by way of example with IBM WebSphere, are complex, monolithic, and focus on high-level process languages such as BPEL. The core process engines used in these solutions are not designed to be used standalone, meaning outside of the corresponding application servers and without the comprehensive set of modelling and code generation tools. Thus, these frameworks

fall short of the important requirements of versatility and generic applicability. With respect to programmability, low-level imperative programming interfaces are either not publicly available or not well documented. Other frameworks such as YAWL or OSWorkflow focus on introducing a new, supposedly more powerful, graphical or declarative process language. While this can make sense on higher levels, for instance, for domain specific tools and languages, a generic core engine should be independent of such notations. It should rather provide an efficient programming model in an easy and well-known language. The jBPM and Windows WF frameworks are the two approaches that, in parallel to this thesis, have developed towards a direction similar to this thesis project. Here, the engine kernel is generic and largely independent of high-level notations and tools. However, these frameworks adopt the classical approach of translating a graphical process model, for instance, Petri nets or state machines, to a corresponding object model in an object oriented programming language. Although this approach maintains tight coupling between the source code of the process and the graphical representation, defining processes using such object models is rather cumbersome. Furthermore, persistence support in these frameworks lack flexibility and thus efficiency during process execution. The process framework presented in this thesis addresses all of these shortcomings. For instance, by introducing a workflow specific programming model rather than a new language or an object model, a flexible persistence model, as well as a lightweight and generic engine architecture that supports the idea of process-driven applications. To the best of our knowledge, no other project exists at the time of writing that focuses on generic workflow kernels in the same distinct way.

The above discussion of the various methods, devices, and schemes for remote authentication and transaction authorization unveils that there is no single solution that satisfies all the requirements presented in Section 1.8.2. Nowadays, resistance against MSW represents the most challenging requirement. Only a few approaches are able to satisfy this requirement in the long term. Standalone challenge/response readers that require the user to input a challenge and the transaction details are examples. Flicker devices as well as Secoder or FINREAD compliant solutions are considered secure as well. However, a solution that does not only ensure security, but also satisfies the additional requirements regarding mobility, convenience, integration, and administration (c.f. Section 1.8.2) is not known at the time of writing. Thus, this thesis introduces a new approach, which uniquely combines all these requirements into an innovative device and method for authentication and transaction authorization. This approach is generically

applicable to all kinds of transactions and supports the multi-party authorization pattern. This makes it relevant as a client device for the process execution framework.

3. A GENERIC FRAMEWORK FOR PROCESS EXECUTION

Having identified the challenges and requirements of a lightweight and generic process execution framework, and after having discussed the limitations of existing approaches, a proposal for a suitable architecture is introduced in this chapter: the *embeddable Process Virtual Machine* (ePVM). ePVM provides a lightweight process execution framework that can be easily embedded into all kinds of applications. It can be considered a micro-kernel type of approach, as it solely focuses on the core, domain independent functionality of a process engine leading to a very generic framework. After introducing the high-level architecture of ePVM, its process model, which is based on the theoretical foundation of communicating extended finite state machines, is presented. Then, it is described how the theoretical model can be applied in a JavaScript-based programming model representing the process definition language of ePVM. Afterwards, it is discussed how the ePVM architecture supports the core functionality of a process engine, namely, concurrency and synchronization, communication, monitoring, and persistence. Finally, the ePVM prototype is described along with a more elaborate example illustrating the use of ePVM to define complex concurrent process flows.

3.1. HIGH LEVEL ARCHITECTURE

The challenges and requirements identified in Chapter 1 along with the limitations of existing approaches identified in Chapter 1 led to the design of a new generic process execution framework named *embeddable Process Virtual Machine* (ePVM). The term “process virtual machine” indicates that ePVM is an interpreter-based approach and that it executes process definitions. However, it is not the definition of a workflow-specific byte code interpreter, as the name may spuriously suggest, but rather a runtime library integrated into an interpreter of a standard programming language. Furthermore, ePVM should not be confused with the Parallel Virtual Machine (PVM) using a similar abbreviation.

Figure 19 shows the top-level architectural design of ePVM including its typical environment. The idea of ePVM can be considered a bottom-up or micro-kernel type of approach for building BPM/workflow systems or process-driven applications. This means that ePVM is a basic framework for building such systems rather than a complete off-the-shelf application that can run standalone. To fulfil the requirement of versatility,

ePVM consists of a self-contained library providing an appropriate API, the so-called *host API*, which allows programmers to easily embed the process engine into another application, the so-called *host application* (c.f. Figure 19). Being self-contained means that all functionality required for basic usage of the library is included. It does not require any server software container to run or any database installation, nor does it have dependencies on other software packages. In case of the prototype implementation, the ePVM library is a Java library such that the host API obviously is a Java API. However, providing ePVM in any other language, for example, as a C library, would not affect process definitions since they use another language. The process definition language, which is JavaScript in the prototype implementation, is independent of the library's environment because ePVM includes a corresponding interpreter as indicated in Figure 19. The use of JavaScript ensures that process definitions can be easily programmed by hand and the learning curve is very low for a large number of people.

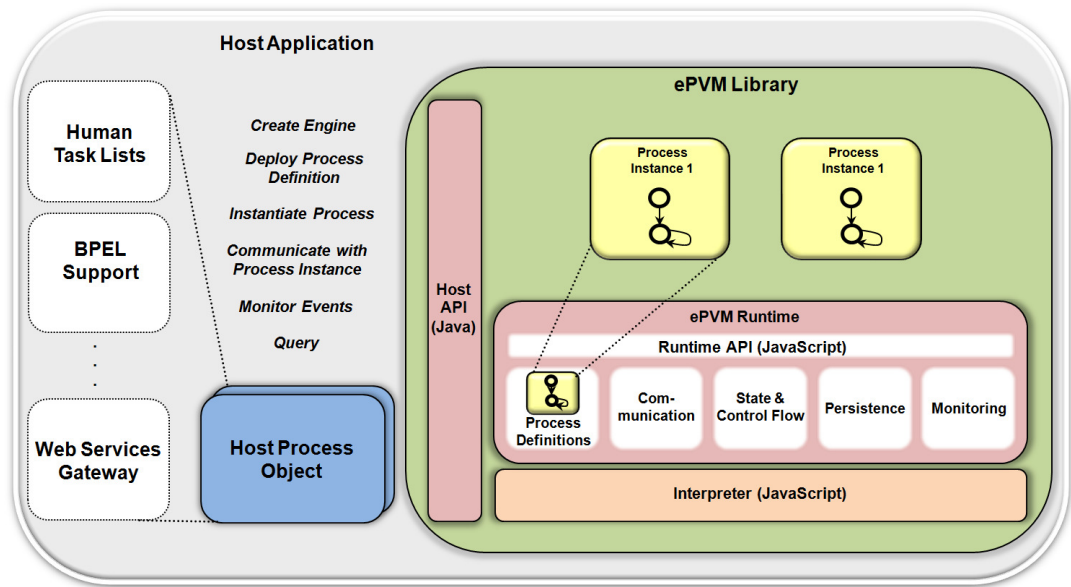


Figure 19. ePVM top level architectural design

The process model of ePVM resembles the theoretical framework of communicating extended finite state machines (CEFSM). However, ePVM does not provide a formal specification language or graphical notation. It rather provides a programming environment and run-time system that explicitly supports structuring process definitions along the CEFSM model. Other process models can be mapped to the state machine model to enable interoperability and support evolution of higher-level process languages. To keep the ePVM engine generic, only the core functionality that constitutes a process

engine is part of the ePVM runtime. The following functionality is considered inherent core functionality (c.f. Figure 19):

- **Process Definitions:** Process definitions can be deployed to the process engine. The engine manages the definitions and supports the creation of one or more process instances from each distinct process definition.
- **Communication:** The process engine provides means for process instances to communicate with other process instances as well as with entities external to the process engine.
- **State and Control Flow:** The engine provides means to manage state data of process instances as well as support for concurrency and synchronization.
- **Persistence:** Process instances can be stored to and retrieved from persistent media as required.
- **Monitoring:** The engine provides means to monitor the execution of process instances as well as the state of the process engine itself.

The ePVM runtime library implements this core functionality and makes it available to process instances via the so-called *runtime API*.

The host application itself communicates with the ePVM engine via the host API. This way, the host application can, for instance, create an instance of the process engine, deploy process definitions, create process instances, or communicate with process instances. Besides the host application, other external entities, so-called *host process objects*, can be registered with the ePVM engine. Once registered, they can communicate with process instances and vice versa. Using this mechanism, higher-level, domain-specific functionality such as Web services support, human interactions or integration with other process languages can be attached on demand. The host process interface represents a general-purpose interface, which can be used to extend ePVM, attach any kind of business code to be triggered from process instances, and integrate the engine with diverse existing infrastructures.

The proposed architecture leads to a generic, lightweight, and easy to program process kernel, which can be embedded into all kinds of applications. For example, it could be used as a page-flow engine within a web server, to implement the application logic of a revision control system, to implement a Web Service orchestration engine, or even for building a fully-fledged BPM system supporting multiple process definition formats and modelling tools. Thus, it addresses the challenges and requirements discussed in Section

1.8.1. Having sketched the top-level architectural design, the rationale behind the chosen architecture and the various design decisions involved are discussed in detail in the following sub-sections.

3.2. PROCESS MODEL

The first decision to be made when designing a process engine is to find a suitable process model – the abstract way to describe processes. There is no standard theory for workflow and business process management that provides a theoretical background like the relational algebra does for databases [93]. Despite efforts of various standardisation bodies, there is no consensus on the representation or conceptual model of a workflow or business process [94]. Thus, there has been a long, and still ongoing, debate about the pros and cons of various formalisms that could be used for modelling business processes. For example, the use of Petri nets [68] versus process algebras such as π -calculus [69] are discussed heavily by the academic community [71].

When looking at the properties of workflows and business processes there are two important perspectives to be considered, the data perspective and the control flow perspective. Van der Aalst [75] and Russell [95] compiled a set of control flow patterns and data patterns that together define comprehensive workflow functionality. Today's trend towards BPM advocates a process-centric modelling paradigm, meaning that the focus is on coordinating tasks and thus on the control flow perspective. In other words, the main challenge in today's process definitions is handling concurrency and synchronisation. In fact, a reasonably complex business process represents a concurrent system. Therefore, the various well-known approaches for the modelling of concurrent systems such as Petri nets, process calculi, or communicating state machines, can potentially be applied to business process modelling. The discussion of related work indicates that Petri nets are the most popular abstraction used in process frameworks. This is the case for several reasons. Firstly, Petri nets are graphical language and thus support communication with end-users, as opposed to process algebras. Secondly, they have a clear and precise definition, because the semantics of the classical Petri net as well as several enhancements such as coloured Petri nets have been formally defined. Lastly, there are tools and techniques available for reasoning on certain qualitative and quantitative properties of Petri nets [93]. However, Petri nets also have some deficiencies concerning workflow modelling. The token game semantics of Petri nets rather describes closed active systems whereas a process engine executing a process definition can be

considered as an open, reactive system. A process engine does not execute the activities within a process, but merely coordinates the execution of these activities by human actors or software components. A process engine does not mainly control its environment, but reacts to events in the environment. Although Petri nets can be extended to that end [96], reactive systems, and this is what workflows are, are usually modelled using event-condition-action rules as used in communicating state machines. The popularity of the Petri net abstraction is probably due to that fact that it is close to flowcharts often used when sketching business processes. However, the current trend towards event-driven and loosely coupled process modelling in the context of SOA requires another abstraction. State machines are well suited for this purpose because they perform actions in response to explicit triggers such as events or messages. In contrast, the flowchart does not need explicit triggers; it rather transitions from node to node in its graph automatically upon completion of an activity. Thus, the way processes are modelled in ePVM resembles the theoretical framework of communicating extended finite state machines (CEFSM) [29,97]. Additionally, whereas Petri net models are usually translated into a corresponding object model for execution (c.f. Section 2.1), the CEFSM abstraction can be easily translated into a corresponding programming paradigm as discussed in Section 3.3. In other words, ePVM does not provide a formal specification language, a graphical notation, or an object model, but rather provides a programming environment and runtime system that explicitly supports structuring process implementations along the CEFSM model. Therefore, ePVM is rather a practitioner's approach although being based on the theory of communicating automata. Process calculi such as the Calculus of Communicating Systems (CCS) or π -calculus could be used to formally describe and verify ePVM processes, but this lies outside the scope of this work. Similarly, the support for graphical representation of process definitions is considered optional. At the level of the core engine, only programmatic (textual) definitions are visible. The ePVM process model is inspired by previous ideas developed in the context of real-time computing, telecommunication systems, and embedded systems [98].

An extended finite state machine (EFSM) is a finite state machine (FSM) augmented by local variables which can be viewed as the complex implicit state within the discrete explicit state of the automaton. Additionally, predicates on the variable values are considered as preconditions for state transitions and operations on the variable values are executed as part of such transitions. This way the state explosion problem known from

ordinary FSMs is dealt with. Furthermore, for each state transition, the values of the local variables are mutually disjoint meaning that the EFSMs are deterministic. Formally, an EFSM M can be defined as follows [98]:

$$M = (Q, \Sigma, \Delta, \underline{x}, T, q_0) \text{ where}$$

Q : is a non-empty finite set of states,

Σ : is a non-empty finite input alphabet,

Δ : is a non-empty finite output alphabet,

$q_0 \in Q$: is the initial state,

\underline{x} : is a finite vector $\underline{x} = (x_1, x_2, \dots, x_k)$ of local variables, and

T : is a non-empty finite set of transitions.

Each transition $\tau \in T$ is a six-tuple $\tau = (q_\tau, \dot{q}_\tau, a_\tau, o_\tau, P_\tau, A_\tau)$ where

$q_\tau \in Q$: is the current state,

$\dot{q}_\tau \in Q$: is the next state,

$a_\tau \in \Sigma$: is the input,

$o_\tau \in \Delta$: is the output,

$P_\tau(\underline{x})$: is a predicate on the current local variable values, and

$A_\tau(\underline{x})$: gives an operation on the variable values.

Whenever the machine receives an input, it evaluates $P(\underline{x})$. Only if this evaluation returns true, the transition τ is executed, which means the machine outputs o , changes the current variable values by $\underline{x} \leftarrow A(\underline{x})$, and moves to state \dot{q} . If the set of variables is empty and all predicates always return true, the EFSM becomes an ordinary deterministic FSM. Concurrent systems can be very well modelled via a number of such EFSMs running in parallel. Input to an EFSM can be queued by some ordering policy and may be generated by other EFSMs resulting in a web of interweaved state machines finally representing the CEFSM model.

The CEFSM concept is well known and often used in the design and implementation of telecommunications systems, network protocols, embedded systems, as well as in hardware design and testing [99,100,101]. It has proven to be a very efficient abstraction whenever event-driven, reactive systems involving many concurrent activities are to be designed. The Specification and Description Language (SDL) [102] is a prominent example that is based on the idea of CEFSMs.

Comparing business processes with these application domains shows that they seem to have a lot in common. For instance, just like a network protocol, a business process can be described on high level as a state machine. In fact, this is what business people often do, consciously or not, when they design a process by drawing circles and arrows on a whiteboard. Furthermore, placing a network protocol into a protocol stack leads to a set of communicating state machines. Again, this corresponds to interacting business processes. Additionally, one protocol is often broken down into a set of so-called micro protocols again consisting of state machines, which introduces some hierarchy of CEFSMs. Complex business processes or workflows in general are very similar to network protocols. They are event driven, reactive systems exhibiting a locally synchronous, but globally asynchronous behaviour just like communicating state machines. Therefore, CESFMs are first class entities in ePVM and are the main abstraction for modelling processes (on micro and macro level), communication, synchronization, and concurrency.

In general, it can be observed that state machine based modelling of business processes is gaining more and more attention at the time of writing. Leading products such as Microsoft's WF or IBM's WepSphere Process Server have been created/updated recently to explicitly support state machines. The reason is twofold. Firstly, event-driven architectures based on messaging-frameworks designed around the enterprise service bus (ESB) pattern have become prevalent as a technology complementing SOA. State machines seem to be ideal for modelling such event-driven applications. Secondly, state machines provide more flexibility than flowchart or activity diagrams in which the flow is usually into one direction and all possible paths within a process are prescribed. However, today's business processes are complex and agile and there are many alternative paths for achieving one goal. Consequently, such processes are easier modelled via state machines because there the idea is not to hardwire all different paths, but rather to account for all possibilities. Furthermore, the CEFSM model provides the basis for statecharts as available within UML. UML statecharts are derived from Harel

statecharts, which add explicit support for hierarchy to CEFSM. UML statecharts could be used as the graphical representation of ePVM process definitions. However, higher-level languages and graphical notations are by design not part of the ePVM framework, as the focus is solely on the core engine functionality and execution. Nevertheless, a mapping from statecharts or other notations such as BPMN to executable ePVM code could well be defined and made available as an ePVM extension.

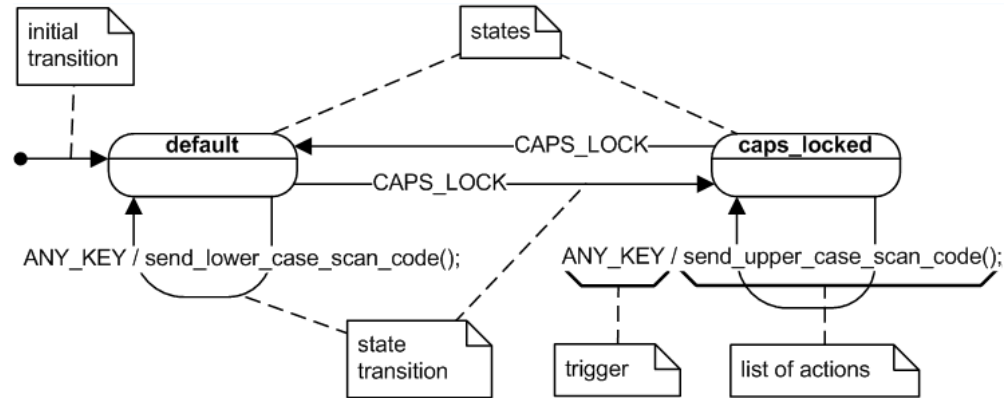


Figure 20. UML state diagram of a computer keyboard state machine [101]

Figure 20 provides an example of an UML statechart, also called state diagram, representing a computer keyboard state machine. State transitions are triggered by events and involve actions. Additionally, UML allows defining so-called orthogonal regions whereas orthogonal means independent. In other words, orthogonal regions allow multiple parts of a state machine to be active concurrently. This corresponds to concurrent state machines in the CEFSM abstraction. In contrast to statecharts, the CEFSM model applied in ePVM does not explicitly support hierarchy. The motivation for that is to keep the engine simple and lightweight focusing only on core functionality in line with the micro kernel paradigm. In this respect, support for hierarchy is considered optional and could be provided as an extension if deemed necessary.

3.3. FROM CEFSMS TO PROCESS DEFINITIONS

3.3.1. Process Definition Language

Having defined CEFSMs to represent the theoretical foundation of ePVM, an appropriate process definition language for defining executable processes is required. In contrast to many other process engines, ePVM does not introduce a new imperative language, a declarative XML-based dialect, or a mix of both. Neither does it provide a mapping of entities of the process model such as state or transition to an object model in an object

oriented programming language as current approaches do (c.f. Section 2.1). Instead, a programming model that allows structuring code along the CEFSM paradigm in a well-known imperative programming language is provided. The JavaScript language has been chosen as the base language and execution platform. Support for implementing state machines is offered via a comprehensive run-time API as indicated in Figure 19, leaving the JavaScript language itself unmodified. Therefore, defining an ePVM process means implementing one or more state machines in JavaScript. Using JavaScript has several striking advantages. Firstly, it is an open and firmly defined language [103] natively supporting XML data types [104]. The latter is not a critical feature, but can be very helpful when operating in a Web service environment. Secondly, it is a very easy, but yet powerful language with various features typical for scripting languages. For instance, it has a simple implicit type system, dynamic garbage collection, and only basic object-based features not making the language too complicated. Thirdly, and possibly most importantly, it has an exceptionally large user base. All this makes ePVM easily accessible for a large number of people. They only need to understand the ePVM programming model and API but do not need to learn a new language. Establishing a new language would be an order of magnitude harder than just establishing a programming model on top of a well-known language. Due to the importance it gained for scripting Web pages, JavaScript is the most used scripting language at the time of writing. Nevertheless, ePVM could also be implemented using any other language with similar properties. In any case, it should be well-known and not too complex. The focus of a workflow definition is not to solve complex problems but only to coordinate tasks, thus easy scripting languages are considered better suited than full blown object oriented programming languages such as Java, or C#. Finally, it is preferable to use an interpreted language to have full control over misbehaving process implementations and be able to easily modify process definitions at runtime.

3.3.2. Process Anatomy

Having decided to use JavaScript as the base language, a programming model resembling the CEFSM approach must be defined along with an appropriate nomenclature. In ePVM, every process is defined by an ordinary JavaScript function referred to as the *process function*. The process function acts as the static implementation of one state machine and as such represents a *process definition*, which can be deployed to the execution engine. Once deployed, it is available for instantiation whereupon an arbitrary

number of instances can be created. A *process instance*, created by *invoking* a process definition, consists of an associated *ePVM thread* and an *environment*. The role of the process function is similar to the role of the code function in traditional threading, however, in ePVM the runtime system calls the process function anew for every message the process instance receives; a message hereby can carry any JavaScript object as payload. The messaging facility provided for communicating with process instances is described in Section 3.5. The process function is similar to the core of an implicit event loop performing a number of actions in response to each received message as outlined in Figure 21. These actions may include sending messages to or receiving messages from other processes, or creating further process instances. Finally, the return value of the process function indicates whether the process instance should remain in the system or has terminated and should be removed. In the former case, the process function will either be called with the next message pending, if any, or falls asleep waiting for a message to arrive.

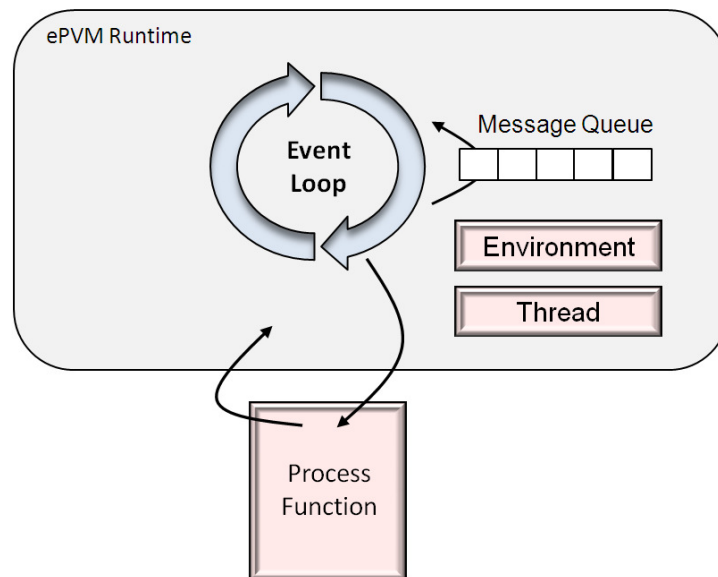


Figure 21. Event-driven invocation of the process function

Listing 7 shows a simple ePVM process named *echo_PVMPProcess*, which simply bounces each message received regardless of its type back to its sender until it receives a stop message of type string and content “*stop*” upon which the process instance terminates.

```

__PVMPackage = "My echo package";

Function echo_PVMPProcess(environment, message) {

```

```

    if (message == "stop")
        return false; // terminate process instance
    PVM_reply(message);
    return true; // keep receiving messages
}

```

Listing 7. A simple ePVM process definition

The ePVM runtime system always calls the process function with the environment and the current message value to be processed as function parameters. Consequently, each process function should expect two parameters to be passed. In relation to the CEFSM model, the process function can be considered as the combined state transition functions of the state machine, whereas the types of messages received or sent resemble the CEFSM's input and output alphabets, respectively. The environment loosely corresponds with the local variables of the CEFSM whereas it may or may not include an indication of its explicit state. Technically, the environment can be any JavaScript object that is initially defined at process instantiation time giving the state machine its initial state. It is up to the user to specify the exact structure of the environment and the exchanged messages and possibly establish conventions. Typically, the environment would include some indication of the discrete state of the process along with supplementary variables. Similarly, messages often consist of a message ID plus some accompanying data. The ePVM framework intentionally leaves these details open to remain as generic as possible. The same applies to how process functions are implemented. All the well-known implementation patterns for state machines are conceivable. For example, one popular technique of implementing state machines is the nested switch statement, with a scalar state variable used as a discriminator in the first level of the switch and the message used in the second. Another popular approach is the use of state tables containing arrays of transition function references for each state. The ePVM framework intentionally leaves all this open and solely focuses on maintaining the environment, messages, and threads. Beside state and control flow management, data flow is another relevant aspect in process frameworks. Russell [95] has investigated data flow patterns in workflow systems and identified the following characteristics as particularly important:

- **Data visibility** – relating to the extent and manner in which data elements can be viewed by various components of a process.
- **Data interaction** – focusing on the manner in which data is communicated.
- **Data transfer** – which consider the means by which data is transferred between processes.

- **Data-based routing** – which characterise the manner in which data elements can influence control-flow.

Concerning visibility, in general, every JavaScript function, given that it is visible in the current JavaScript scope, can either be called via an ordinary function call or used to create an ePVM process instance. The JavaScript language provides only a few mechanisms to group functionality and to restrict access. In particular, all scripts are normally executed under one shared top-level scope called the global scope. To allow additional means for structuring process definitions, ePVM introduces the notion of *process packages*. A process package is a piece of JavaScript source code containing one or more related process definitions. All process instances invoked from the same process package share one package-level scope, yet there is no shared global-scope between packages. Process definitions can only be deployed as part of a process package, which is identified by a unique package name. Process functions tagged with the specific postfix `__PVMProcess` are accessible by name from other packages and from the host application; others can only be accessed via its function reference either visible within the package or explicitly passed between package scopes. Listing 7 shows the definition of a process package named “My echo package” including one externally visible process definition named `echo_PVMProcess`. As object references can be freely passed between processes, it is up to the programmer to decide the extent to which the concept of process packages is exploited for strict code or data separation. Process instances can share data in various ways. For instance, through references stored within the environment, global variables within the package scope, or via references passed within messages. Although there is no global JavaScript scope across process packages, ePVM also supports globally shared data via a dedicated object, named *PVM_global*, which is visible in all package scopes. ePVM leaves it to the users to decide on which level, package local or global, processes make use of shared memory. Alternatively, processes may rely on message passing not using any shared memory at all. In other words, all the different data flow patterns can be implemented with ePVM.

Concerning the data interaction characteristics, the prime mechanism is the messaging facility discussed in detail in Section 3.5. It is designed to be uniform across all participating entities. This means that there is no difference between inter-process messaging and messaging between an ePVM process and an external entity, namely a host process object.

Concerning data transfer, the passing of data is in accordance with the JavaScript specification. This means that primitive values such as numbers and strings are passed by value whereas JavaScript objects are passed by reference. However, this does only apply for data flow within the process engine. In case of communication with external entities all data is passed by value to prevent synchronization issues due to preemptive threading as discussed in Section 3.4 and 3.5.

Concerning the data-based routing characteristics, the control flow within ePVM is inherently message-driven and all kinds of data such as message values, environment values, or shared memory can be considered for driving control flow by sending messages.

```
function ProcessPackage(scope, name) {
    this.scope = scope;
    this.name = name; // package name (unique per engine)
    this.monitor = false; // package monitor object registered?
    this.monitorEvents = 0; // events to be monitored
    this.monitorMode = 0; // 0 -> asynch; 1 -> synch
    this.useCount = 0; // #existing instances (process or handler)
}

function Process(processPackage, functionName, functionObject,
    environment, type, handle) {
    this.pkg = processPackage;
    this.funcName = functionName;
    this.funcObj = functionObject;
    this.environment = environment;
    this.handle = handle;
    this.thread = null; // thread executing this process
    this.inQueue = new InQueue(); // incoming messages
    this.saveQueue = new SaveQueue(); // saved messages
    this.currentMessage = null;
    this.type = type; // PROCESS/HANDLER
    this.persistenceProvider = null;
}

```

Listing 8. ePVM internal process and package anatomy

Listing 8 shows how process packages and process instances are structured internally in ePVM by means of JavaScript object constructor functions. Essentially, a process package represents a JavaScript scope including one or more process definitions and it is identified by a unique name. Other fields regarding monitoring support are discussed later in this chapter. A process instance belongs to a process package, references its process function, and contains the environment holding its state. Furthermore, each process instance is identified by a unique handle. The other fields are related to messaging, threading, and persistence as discussed later in this chapter. Package names, process function names, and process handles are the only information that is relevant for

ePVM users. This information is used in the ePVM APIs to create instances and communicate with them.

3.4. CONCURRENCY AND SYNCHRONIZATION

3.4.1. ePVM Threads

As mentioned, ePVM process instances, just like communicating state machines, run in parallel and therefore each process instance is associated with one thread. Since JavaScript does not natively support multi-threading, the threads used for this purpose are provided by the ePVM runtime system. Such threads are referred to as *ePVM threads* and are distinct to threads provided by the underlying operating system (OS) referred to as *OS threads*. The threading system implemented in ePVM can be considered an event-driven cooperative user-level threading based on continuations. With continuations, the JavaScript interpreter allows the execution of scripts to be suspended and resumed while maintaining their complete execution context. ePVM exploits this feature to introduce ePVM threads while the ePVM engine only runs within one OS thread. The threading system is event-driven in a sense that the scheduling of threads depends on messages received and sent by its associated process instance. For example, sending a message to a process instance causes its thread to become subject to scheduling. Listing 9 provides a very simple example illustrating this via two interacting ePVM process instances. It is assumed that a process instance of *ProcessA* exists and a message has been sent to it. Consequently, then the ePVM runtime system invokes the process function for processing the message. Firstly, *ProcessA* creates an instance of *ProcessB* via the ePVM runtime API function *PVM_invoke()* (c.f. Listing 9). At this point ePVM internal structures (c.f. Listing 8) representing the process instance are created including an ePVM thread. However, the thread is not scheduled yet. Secondly, *ProcessA* sends a message to *ProcessB* via the API function *PVM_call()*. This function sends the message synchronously, meaning that it is blocking until *ProcessB* sends a reply message. Consequently, *ProcessA* falls asleep and its thread is no longer considered for scheduling. Since *ProcessB* now has a pending message, the ePVM runtime considers the related thread for scheduling. The ePVM internal scheduler uses a simple priority based scheduling strategy where threads can have a priority between 1 (low) and 10 (high). The priority can be defined when a process instance is created. The last parameter of the *PVM_invoke()* function specifies the priority, which is 1 in case of *ProcessB*. As soon as *ProcessB* is triggered with the message from *ProcessA* it simply bounces the message

back to *ProcessA* via the *PVM_relp()* API function and then it terminates. As a result, *ProcessA* is again considered for scheduling since it has a pending message. Finally, once *ProcessA* is scheduled, the *PVM_call()* function returns with the reply message and *ProcessA* simply terminates. This gives a first impression of how concurrency and synchronisation are implemented in ePVM. Details about the messaging facility and the ePVM APIs are provided later in this chapter.

```
function PcessA__PVMProcess(environment, message) {
    // create instance of ProcessB
    var hnd = PVM_invoke(ProcessB, PVM_PROCESS, null, 1);
    // send a message synchronously
    var replyMsg = PVM_call(hnd, null);
    return false; // terminate
}

function PcessB__PVMProcess(environment, message) {
    PVM_reply(message); // bounce message
    return false; // terminate
}
```

Listing 9. Event-driven threading example

The ePVM threading system implements a so-called cooperative threading model meaning that threads cannot be interrupted at any point, as opposed to preemptive threading used with OS threads, but threads themselves relinquish control for other threads to be scheduled. For this model to work properly, message processing within process functions must be non-blocking and short-lived to ensure that the process engine remains responsive. Therefore, the following rules must be followed when implementing process definitions:

1. Message processing must be short-lived. The main purpose of process definitions should be coordinating tasks and thus message processing is typically short-lived by nature. If message processing requires significant computing, for example, if decisions to be made involve complex algorithms, the processing must be off-loaded to host process objects, since they run in separate OS threads. The concept of host processes is described later in this chapter.
2. Blocking behaviour should be implemented via the corresponding ePVM API functions. For example, via the *PVM_call()* function as used in the example shown in Listing 9. The implementation of these API functions automatically yields control if required. The ePVM runtime API documentation describes which API functions might yield control.

Using an event-driven cooperative threading system within ePVM has a number of advantages. Firstly, it reflects the nature of business processes and workflows very well.

Business processes are often ‘long-running’ processes, which are idle most of the time. They are meant to coordinate tasks, human or computer, and not to execute tasks themselves. Therefore, their behaviour is rather reactive than active. For example, a process instance might put a task in the task list of an employee and then wait for the employee to report the task as completed. Only when the completed notification is received, the process instance becomes active again, performs some processing, possibly changes its state, and triggers another operation. The processing within the process instance is usually limited to deciding what is to be done next and to triggering some external entity to do the ‘real work’. Secondly, the fact that process instances always run exclusively and cannot be preempted arbitrarily but only with well-defined system calls, significantly eases synchronization of shared-memory access. No low-level and error-prone synchronization primitives such as semaphores are required. Instead, synchronization can be solely implemented via message passing, leading to an easy and consistent way of programming concurrent processes.

One drawback of the approach is that the responsiveness of the system relies on process instances being cooperative. Therefore, it might happen that an erroneous process definition, for example, a definition including an endless loop, blocks execution of the complete process engine. However, such situations can be handled by limiting the maximum time between context switches within the underlying JavaScript interpreter such that misbehaving process instances can be detected and removed from the system. Furthermore, the process engine itself cannot take advantage of multi-core platforms due to the user-level threading. At first sight, this seems to be a limitation considering that multi-core platforms are becoming prevalent at the time of writing. However, considering the fact that processes are idle most of the time, a single threaded execution engine is appropriate. Assigning one OS thread to every process instance is not necessary and might even decrease overall performance due to additional scheduling/memory overhead and more complex synchronization constructs that would be required. Furthermore, it is important to note that the single threaded nature only applies to the core execution engine. Multi threading is required heavily on the boundary to the host application, for the execution of host process objects. Considering that processes only coordinate the work, and the actual work be it external communication or computation is done by entities external to the process engine, justifies the design. In other words, dedicating only one CPU core to the execution of process definitions and all others to the execution of the ‘real work’ matches the nature of process definitions. Nevertheless, large-scale

BPM application scenarios might still create scalability issues requiring multiple process engines to be installed on distributed resources. In such cases, it is assumed that scalability can be achieved on application level rather than on process engine level. This means, external requests triggering the instantiation of a new process instances, for instance, an incoming order triggering a purchase order process, are distributed among a number of process engines via some load balancing entity. Alternatively, the framework could be extended to allow process instances to travel between process engines. However, this would make the framework much more complicated and would restrict flexibility, for instance, in the area of shared memory and reference passing.

3.4.2. Process Types

It is vital for the performance of the user-level threading system in ePVM that the underlying interpreter is optimized for fast context switches. However, it still makes sense to eliminate context switches whenever possible. For this reason, ePVM introduces process instances of type *HANDLER* in addition to the normal process instances, of type *PROCESS*, as described so far. A process instance of type *HANDLER* (a handler, in short) is not statically associated with one ePVM thread. Consequently, if a message is sent to the handler, it simply borrows the thread of the sending process effectively eliminating two context switches. If the handler is busy because it is currently processing a message from another process, the new message gets queued. Pending messages are then processed by the thread of the process for whose message the handler was actually called. The restriction is that handlers must not use blocking system calls, for instance, sending messages synchronously. Blocking calls would block the borrowed thread and thus the corresponding process instance, which could lead to deadlocks. In general, for a process instance sending a message to another process instance, it should not make a difference whether the target process is of type *HANDLER* or not. Apart from excluding blocking behaviour, there is no difference between the implementation of a handler and a normal process. The decision which type of process instance is created is solely made at instantiation time. Handlers might also be used for synchronization purposes as they match the notion of monitors if the messages they process have been sent synchronously.

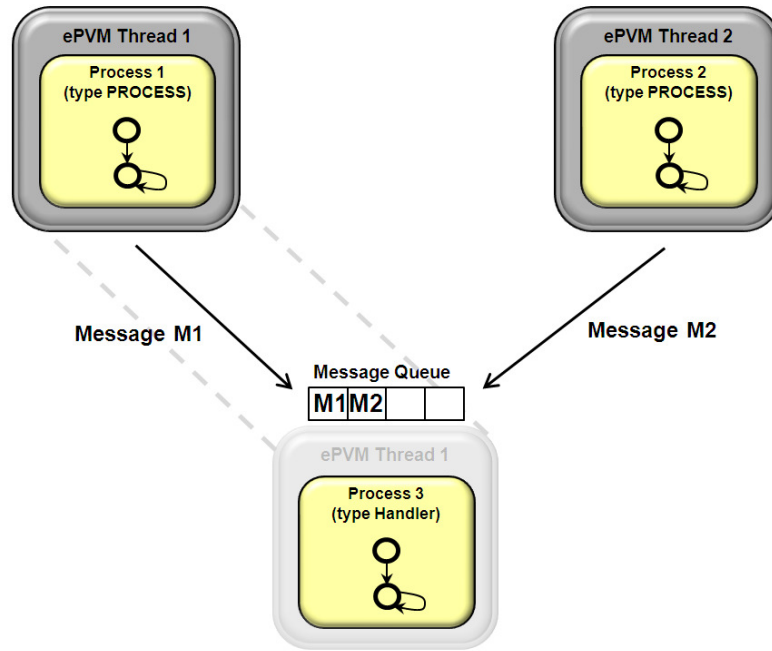


Figure 22. An ePVM process instance of type *HANDLER*

Figure 22 illustrates how a handler process works. Assume that process instance 1, which is of type *PROCESS* and thus has an associated ePVM thread, sends message M1 to process instance 3, which is of type *HANDLER*. Furthermore, it is assumed that the handler is idle at that point in time meaning that it does not process a message. In this case, the ePVM runtime system would immediately trigger the process function of process 3 with M1 within the execution context of process 1. Effectively, this means that process 3 is borrowing thread 1 from process 1 for processing M1. Beyond that, let us assume that process 2 sends message M2 to process 3, while process 3 is still busy processing M1. In such case, M2 is queued until M1 has been processed. Afterwards, process 3 processes M2 while still using the borrowed thread from process 1. In other words, a handler, if idle, borrows a thread when it receives a first message and then uses it until it becomes idle again, meaning that it has no more pending message. In general, handlers can be considered lightweight process instances, which can be used to structure a process flow into multiple state machines without requiring additional threads and thus system resources. In addition, minimizing context switches obviously increases execution performance. Another restriction introduced by handlers is the fact that every entity sending a message to a handler must be in possession of an ePVM thread. Within ePVM this is always the case, for example, process instance 1 in Figure 22 might also be a handler in possession of a borrowed thread. However, external entities such as host

process objects, introduced in Section 3.5.2, do not have an ePVM thread and thus cannot communicate with handlers.

The example provided by Listing 9 can be consulted to illustrate the use of handlers. In this example, *ProcessA* creates an instance of *ProcessB* via the *PVM_invoke()* API function. Simply changing the second parameter in this function call from *PVM_PROCESS* to *PVM_HANDLER* causes a process instance of type *HANDLER* to be created. As a result, the two context switches required for switching between the two process instances during the synchronous message exchange are eliminated. Apart from that, the process semantics remains unchanged.

In addition to the concept of handlers, ePVM provides the possibility that multiple process instances of type *PROCESS* can share one ePVM thread and thus form a so-called *process group*. More precisely, a normal process instance is just a special case of a process group where the number of processes in the group is one. A process group introduces a serialization mechanism on the level of message processing. As the thread is shared between members of a group, only one process function can execute at any point in time. This process function must return before any other process function within the same process group can be triggered. This way it is enforced that processes within a process group do not process messages concurrently. Process groups can be a very helpful construct in cases where multiple process instances are working on one globally shared state and they should not execute in parallel by design.

3.5. COMMUNICATION

3.5.1. Inter-Process Communication

Inter-process communication targets the communication between ePVM process instances. In accordance with the CEFSM model, ePVM provides a message-passing facility for this purpose. Each process instance is associated with one message queue storing messages received but not yet processed. This queue is referred to as the *in-queue* of the process. By default, its organization is first-in-first-out (FIFO) but other user defined algorithms are possible. In general, a message object within ePVM consists of the message value and additional context information, for example, a reference to the sending process. The message value can be any JavaScript object, which means that the structure and semantics are completely application specific. Basic types such as numbers or strings might be passed as well as references to complex objects. When the ePVM runtime calls a process function, one message is removed from the in-queue and its value

is passed to the process function; this message thereby becomes the so-called *current message* (CM) of the process. Related context information remains internal to the run-time system, whereas some information such as the message sender can be retrieved via the run-time API. Although in principle the process function is called anew for each message, it is alternatively also possible to explicitly receive the next message from the in-queue via the API. In this case the CM is released, the next message from the in-queue becomes the CM, and the API function returns the message value. Process instances are identified by unique *process handles*, which are used for addressing a particular process instance when sending a message. Messages can either be sent asynchronously if a reply is not needed immediately or not required at all, or synchronously if there is nothing to do for the sending process until a reply is received. In the first case the message is simply put into the in-queue of the target process and the *send* operation returns immediately. The latter case is referred to as a *call* operation, indicating that the sender is blocked until a reply is sent. A reply is identified as corresponding to a call operation by using the same message object. As a result, it is possible to forward a message to a third process, which then can respond to the original sender by replying to that message. If a call message is released without sending a reply, the blocked caller is notified via an exception.

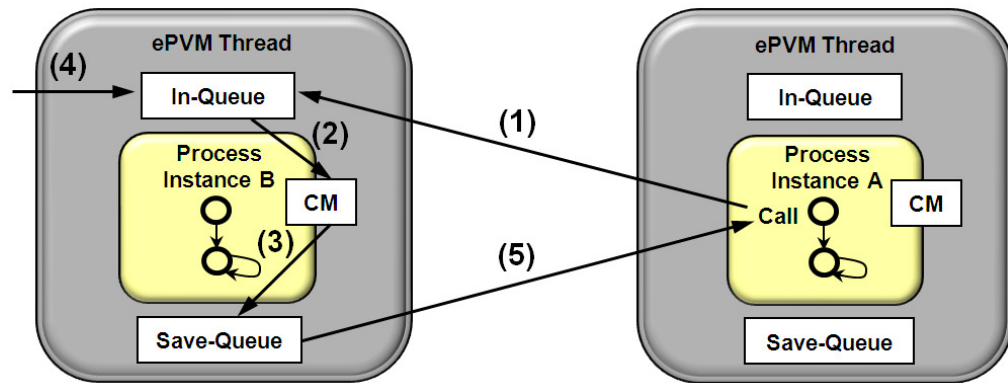


Figure 23. Inter-process communication

In complex scenarios with many concurrent ePVM processes, it happens frequently that a process receives a message that it cannot completely process immediately due, for example, to an outstanding message from another process. To handle such cases, an additional message queue is associated with each process instance, the so-called *save-queue*. Saving a message means moving the message object representing the CM to the save-queue. Figure 3 illustrates this scenario. Process A sends a synchronous message (call operation) to process B and gets blocked (1). Then B is activated and the message

becomes the CM (2). B, however, is missing some information and therefore does not reply immediately but saves the message in its save-queue (3). At some later point, after eventually receiving the information required to answer A's request (4), B wakes up A by sending a reply to the message stored earlier in its save-queue (5).

Lets have a look at a real-world example that corresponds to the scenario illustrated in Figure 23. The source code of the example is shown in Listing 10 providing the definition of two processes, *verifyOrder* (A) and *cardValidation* (B), corresponding to process A and B in Figure 23. Process A initially receives an order message including at least a customer name and credit card number. It then invokes B and sends a synchronous message including these details. B expects the host process named 'cardService' to be available and forwards the message to it asynchronously. Furthermore, it saves A's message to its save-queue, updates it state, and finishes message processing. Assume that at some later point in time, the host process sends a response message (value of type boolean) to B. Then, B simply forwards this message to A by sending a reply to the previously saved message. This reply wakes up A, which then replies with an appropriate result string to the initial message. This example demonstrates the simplicity of defining interacting concurrent processes in ePVM.

```
__PVMPackage = "Order Management";

function verifyOrder__PVMProcess (env, order) {
    // prepare message
    var msg = {name:order.name, number:order.creditCardNumber};
    var handle = PVM_invoke(cardValidation); // create process instance
    msg = PVM_call(handle, msg); // send synchronous msg
    PVM_reply(msg?"accepted":"rejected"); // reply result string
    return false; // terminate process
}

function cardValidation(env, msg) {
    if (env.state == null) { // initial state
        // send msg asynchronous
        PVM_send(PVM_hostProcess("cardService"), msg);
        PVM_save(); // put current-message into save-queue
        env.state = "requested"; // update state
        return true; // wait for next message
    }
    // forward boolean message from cardService by
    // replying to saved message
    PVM_replySaved(msg);
    return false; // terminate process
}

```

Listing 10. Order management example process

By default, the save-queue is organized last-in-first-out (LIFO) but it is possible to register a user-defined compare function, which is then used to sort the save-queue

arbitrarily. In the latter case, the queue is sorted immediately when the compare function is registered and whenever a new message is saved. The compare function can be overridden at any point in time. Furthermore, the runtime API allows searching for messages in the save-queue for reading or removing them from the save-queue, forwarding messages from the save-queue to other processes, or sending a reply to a particular message in the save-queue. The save-queue can be considered part of the local variables of the EFSM whereas in some cases, for instance, where message objects are no longer required for sending a reply, it is up to the programmer whether to keep information in the environment or in the save-queue. Releasing a message finally means that the process has finished processing it and the message object can be destroyed or reused internally. This implies that it is no longer possible to send a reply to, save, or forward the message. The ePVM runtime releases messages automatically upon one of the following actions:

- the process returns from the process function and CM is not null;
- the process explicitly receives the next message and CM is not null;
- the process removes a message from the save-queue.

In addition, the CM becomes null if the message object is saved in the save-queue, forwarded to another process, or if a reply is sent. This implies that these operations can only be applied once to a message.

3.5.2. ePVM-Host Communication

As mentioned, the ultimate purpose of ePVM processes is to coordinate activities rather than execute them. Therefore, an integral part of process execution is the triggering of external entities, which do the ‘real work’ such as sending an email, invoking a Web service, or accessing a database. As an ePVM engine never runs standalone but is always embedded in a host application, ePVM-host communication is the only way for the engine to interact with its environment. The interface between ePVM and the host application is the host API as shown in Figure 19. In the current research prototype, it consists of a Java API but in general, it might be available in other languages, too. This has no impact on the runtime API so that process definitions remain platform independent. All communication from the host application towards the ePVM runtime and to a particular ePVM process instance goes through the host API (1) as indicated in Figure 24. It allows the host to create instances of the ePVM engine itself, deploy process packages, instantiate processes, send messages to process instances (synchronously or

asynchronously), and so forth. Since the host application and the ePVM runtime system run in different OS threads, communication is internal via the *host queue* as shown in Figure 24. The ePVM runtime has a built-in process, the *host device handler*, which is scheduled regularly to process messages from the host queue (2). It either distributes messages to process instances (3) or processes them directly if targeted to the runtime itself. If the host application sends a message synchronously, the host thread will be blocked until a process instance sends a reply to that message or the message is released. Conversely, it is also possible for an ePVM process instance to initiate communication towards the host. For this purpose the host application can register so-called *host processes objects* (host processes, in short), each registered under a unique name and associated with a process handle just like a process instance. Consequently, a host process can be addressed and used for message passing just like a process instance. However, it is not under the control of the ePVM runtime, meaning that it has no associated in-queue, save-queue, or environment and it cannot be instantiated. A host process can be considered a static process or message handler whereas it is up to the programmer whether it maintains its own state or not. When a process instance sends a message to a host process, the ePVM runtime employs a dedicated thread from a thread pool (4) to deliver the message to the host process (5). This thread pool consists of OS threads as opposed to ePVM threads. This means that the host process can be called from multiple threads concurrently, one for each message to be delivered. When processing a message, the host process can use the host API (6) to further interact with the ePVM runtime and its process instances as described for the host application previously. Additionally, it can send a reply to the message it is currently processing. This way ePVM-host communication can be either asynchronous or synchronous in both directions. In principle, the host application could communicate with a host process via the ePVM engine, however, this should be avoided as it is not very efficient and ePVM is not designed to act as a messaging backbone. The intention is rather the reverse, for instance, connecting ePVM as an orchestration engine to an enterprise service bus.

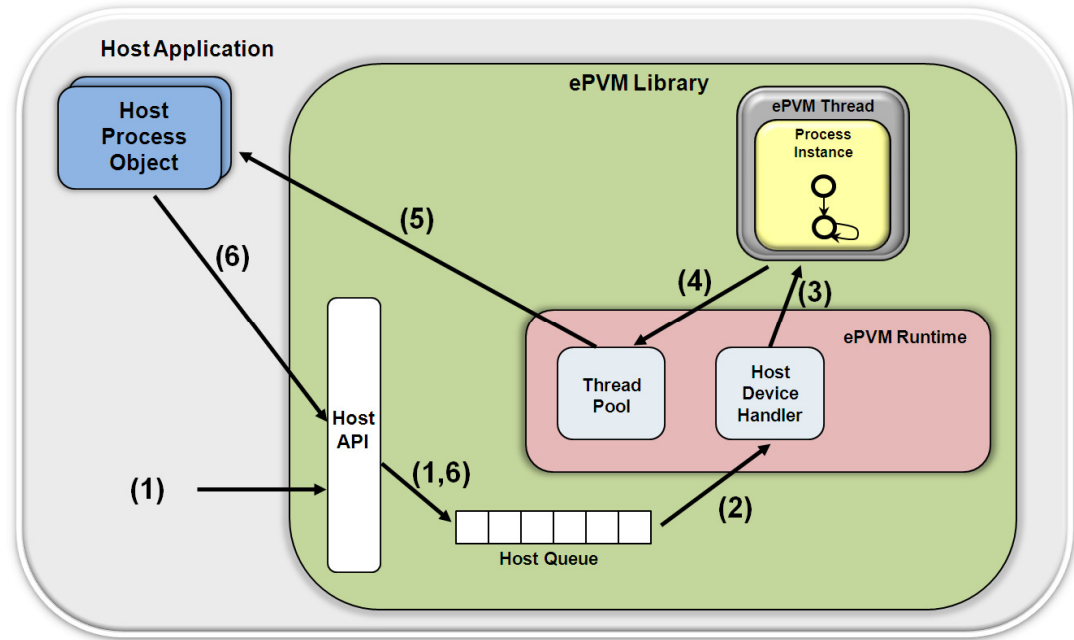


Figure 24. ePVM-host communication

Messages exchanged between ePVM and the host application effectively are JavaScript objects. In the current prototype implementation, the runtime automatically converts basic Java objects such as *String*, *Number* and *Boolean* to the corresponding basic JavaScript types and vice versa. Complex JavaScript objects such as *Object*, *Array*, or *XML* can be constructed and accessed via the host API. The structure and semantics of messages are application-specific and are not defined by ePVM. Section 3.8.5 provides an elaborate example demonstrating the use of host process objects and the ePVM APIs in general. Nevertheless, Listing 11 shows the source code of the host application as well as the implementation of the host process object named *cardService*, which drives the process definition shown in Listing 10. Registering a host process object with the ePVM engine is simply done via the *registerHostProcess()* method providing the host process object and a unique name. The host process object must implement the *PVMHostProcess* interface consisting of the *handleMessage()* method. This method is called by the ePVM runtime system in a dedicated OS thread whenever a message is sent to the host process. Listing 10 and Listing 11 together already represent a fully functional process-driven application based on the ePVM framework.

```

...
// create ePVM engine
ProcessVM pvm = new ProcessVM(null, 0);
pvm.start(); // start engine
// deploy process definition available in the src string
String pkgName = pvm.deploy(src, "", null);

```

```
// register host process cardService
pvm.registerHostProcess(new CreditCardService(), "cardService");
// create instance of verifyOrder process
String instance = pvm.invoke(pkgName, "verifyOrder");
// build JavaScript message object
PVMObject pvmObj = pvm.buildObject("{\"name\":\"John Doe\",
\"creditCardNumber\":\"1234123412341234\"}");
// send message synchronously
String res = (String)pvm.call(instance, pvmObj);
// print process result
System.out.println("Order has been " + res);
// stop ePVM engine
pvm.stop();

// the host process object implementation
public class CreditCardService implements PVMHostProcess {

    public void handleMessage(ProcessVM pvm, String senderProcess,
                               Object message) {
        PVMObject msg = (PVMObject)message;
        System.out.println("Verifying card: " +
                           msg.getProperty("number"));
        pvm.reply(new Boolean(true)); // card valid
    }
}
```

Listing 11. Host application/process example

3.6. MONITORING

Process monitoring is the ability to track process instances throughout their life-cycle. It is the basis for reporting, audit-trail generation, logging, and troubleshooting. ePVM provides means to implement process monitoring on various levels of granularity. For this purpose, the host application can register so-called *process monitors* on the level of process packages. This means a given process monitor, if registered, receives monitoring events of all process instances created from the process package for which the monitor is registered. Once registered, monitors are called via dedicated OS threads, similar to the way that host processes are called, whenever the ePVM runtime issues a monitor event. Furthermore, it can be specified at the time of registering in which types of events the process monitor is interested. A monitoring event consists of the following information: the event type, the process handle and package name, the date, and some supplementary information that varies depending on the event type. The ePVM runtime supports the following basic event types documenting the life-cycle of a process instance:

- **CREATED_EVENT:** A process instance has been created. The supplementary information provides the handle of the creating process.
- **RUNNING_EVENT:** The process function is invoked with a pending message.

- **CALLING_EVENT**: The process sends a synchronous message to another process (call operation). The supplementary information provides the handle of the target process.
- **RECEIVING_EVENT**: The process explicitly receives a message via the `PVM_receive()` API.
- **WAITING_EVENT**: The process has returned from its process function and the in-queue is empty.
- **TERMINATED_EVENT**: The process has terminated normally.
- **FAILED_EVENT**: The process has terminated abnormally due to an unhandled exception. The supplementary information provides a description of the exception.

These events can be considered the basic state machine events for which a monitor can register. Additionally, a process can issue custom events (type `CUSTOM_EVENT`) via the runtime API at any point while executing. This allows the process designer to decide to which level of detail a process is traceable. For instance, a process can fire a custom event reporting its high-level state to its monitor only if important messages are processed. Alternatively, multiple events might be fired reporting all kinds of detailed information about the process state in the course of processing a single message. The level of monitoring also depends on how processes have been modelled within the CEFSM model. For example, one could model a simple decision within a workflow as a single ePVM process in order to achieve extremely fine-grained monitoring.

Like with ePVM-to-host communication, the ePVM runtime can call monitors either synchronously or asynchronously depending on how they have been registered. In the former case, the process instance firing the event is suspended until the monitor has finished processing it. In the latter case, the process instance continues its processing immediately. The monitoring interface can be exploited to add all kinds of related functionality to ePVM. For example, logging process history into a database, providing real-time monitoring, maintaining a related graphical process representation, or applying business intelligence tools. Section 3.8.5 provides an elaborate example demonstrating the use of process monitors for real-time monitoring.

3.7. PERSISTENCE

3.7.1. Introduction

By default, process engines, as well as the process instances they execute, reside in transient memory without any kind of persistent state maintenance. For short-lived and very active processes, this may actually be sufficient in many application scenarios. However, there are a number of scenarios that require a mechanism for persisting state information to a durable medium for different reasons:

- **Resource consumption:** So-called long-running processes represent time-consuming tasks which trigger long-running activities that may take minutes, hours, or even months. Such processes are idle most of the time while waiting for an external trigger and, therefore, always holding them in transient memory does not make sense. It is rather desirable to remove such process instances from transient memory while they are idle and store them persistently to minimize resource consumption. This way, the process engine will be able to handle a larger number of concurrent process instances considering that many of them are idle at any given point in time.
- **Process durability:** In some cases, it is desirable that process instances survive application restarts. For example, if it is required to shut down the process engine for restarting the machine on which it is running on. In this case, the state of the process engine itself may need to be stored persistently in addition to the process instances. To allow this, the process engine must be stopped in a controlled way, as process durability does not necessarily imply crash resilience.
- **Audit trails and transactions:** If a history of persistent process states is kept, they represent an audit trail and it is possible to restore such states to implement local transactions.

For persistence support in the ePVM framework the focus is on the two basic and most important applications of process persistence: resource consumption and process durability. Support for transactional workflows [105] is considered for future work, as it represents an extension to the basic persistence model presented in this section.

Most contemporary process engines provide support for process persistence. However, the implementations lack flexibility in many respects (c.f. Section 2.1). Firstly, it is not possible to attach multiple persistence providers, which allow storing process states to different media such as files or databases. Secondly, oftentimes it is not easily possible to

dynamically define if a process state should be stored persistently and to which persistent store. For instance, for organizational or performance reasons it can make sense not to store all processes into one database or to store a process into a different database as soon as it has reached a certain state. Additionally, it should be possible to define custom persistence strategies such as storing a process persistently after it has been idle for a certain time or if a certain combination of events has happened.

Not all engines provide an easy way to implement such strategies for individual processes, if possible at all. This is due to the fact that most process languages such as BPEL do not consider persistence properties and thus it is assumed that persistence support should be transparent for process definitions. However, fine grained persistence control can be very important not only to implement certain semantics but also to optimize performance and resource consumption. For example, making sure to store processes persistently only if absolutely necessary, and not simply for every single wait state, can increase system performance since persistence operations are typically time consuming. Vice versa, removing processes from transient memory as soon as it can be anticipated that they will be idle for a rather long time frees system resources. Therefore, the model implemented in ePVM does not define persistence policies statically and on engine level, but enables dynamic and fine-grained control and also supports attaching multiple pluggable persistence providers to the process engine.

3.7.2. A flexible Persistence Model

The persistence model of ePVM is generically applicable to state machine based workflow systems. Therefore, this section describes it independently of ePVM. The ePVM specific considerations are discussed in Section 3.7.3. The model assumes that business processes are defined by one or more state machines which run in parallel, meaning that they represent independent activities. The state machines, identified through unique identifiers, communicate by exchanging messages via a communication facility provided by the process engine. Furthermore, a state machine is considered idle if it has no pending message available. In such an environment a state machine typically consists of the following items:

- Its definition;
- its state, that is, its state variables and possibly communication queues; and
- the activity executing the state machine, that is, the associated thread.

A state-machine definition, in this context also called process definition, is deployed to the process engine only once and used afterwards by the process engine to create one or more state machine instances sharing the same state-machine definition. Thus, the state-machine definition does not have to be part of every instance's state, but is considered to be part of the process engine's state instead. Consequently, state-machine definitions only need to be stored persistently if the process engine is to be stopped. For state-machine instances, also called process instances, then only their state and thread must be stored persistently. This is done by serializing the information and sending it to a so-called persistence provider, which is attached to the process engine. The persistence provider, in turn, stores the information to a durable medium. Conversely, when a process instance is to be restored, its information is retrieved from the persistence provider and the instance is reconstructed in memory.

Assuming such a runtime environment, the basic ideas for a pluggable and selective persistence model that leads to flexibility in defining persistence behaviour for process instances are:

1. An arbitrary number of persistence providers can be attached to the process engine and each provider can be uniquely addressed.
2. Every time a process instance processes a message it can dynamically decide if it should be stored persistently and, if yes, by which persistence provider in case it should become idle upon finishing message processing.
3. The process engine can serialize the process state and the associated thread, and forward it to some persistence provider.
4. Any process can request that some other process is stored persistently via a given persistence provider if the process in question is idle.
5. The process engine automatically restores a process stored persistently as soon as it has a pending message for that process.

Figure 25 provides a conceptual overview and example flow illustrating how the persistence model works. In step 1 (c.f. numbers in Figure 25) it is assumed that process instance A sends a message to process instance B. Due to the fact that communication always is indirect via the process-engine runtime, the engine can take additional action at this point. Steps initiated by the engine runtime are denoted by dashed arrows in Figure 25. The engine checks, in step 2, if the target process B is currently stored persistently and, if yes, by which persistence provider. This information is held by the engine runtime in the persistence table. Since process B is stored persistently, the engine retrieves it

from persistence provider P1 and reconstructs the process instance in memory in step 3. Finally, the message is delivered to B in step 4.

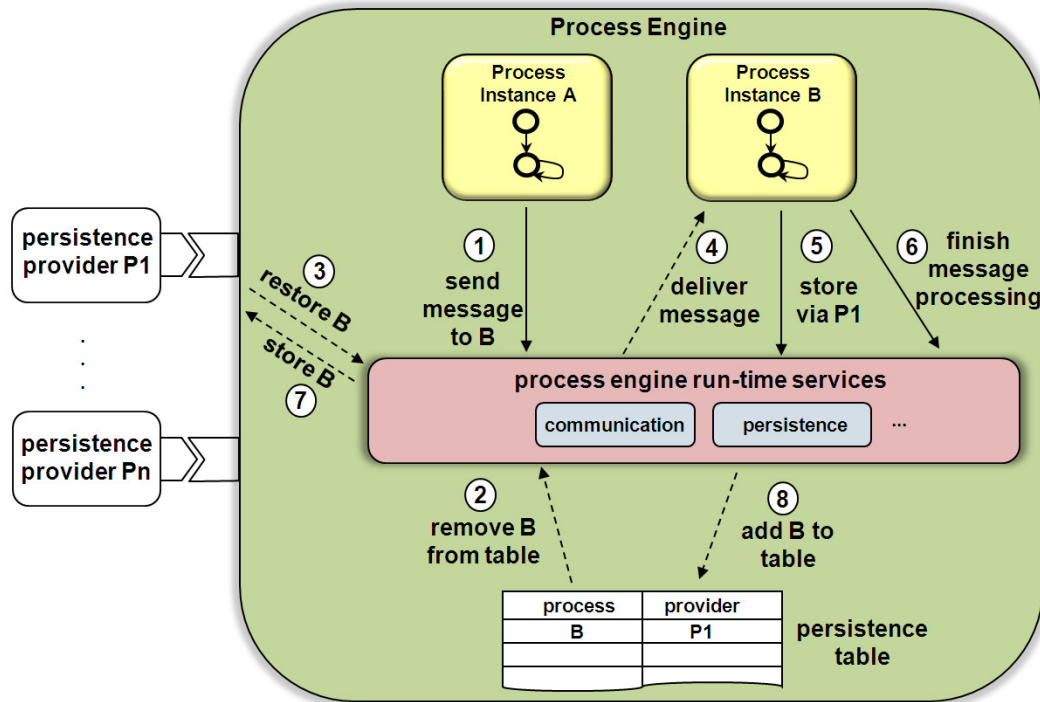


Figure 25. Conceptual overview and example flow

While process instance B is processing the message, it can indicate (step 5) to the process engine that it should be stored persistently via persistence provider P1 if it becomes idle after finishing message processing. As soon as this happens in step 6, the process engine serializes process instance B and stores it persistently via P1 as indicated by step 7. Finally, the persistence table is updated in step 8. This is how process instances get stored persistently and restored again in the proposed persistence model. From a process perspective, it is very easy to precisely control its own persistence behaviour on the level of individual messages, while the actual serialization and restoration is handled transparently by the process engine.

The model does not assume any explicit relation between processes such as a hierarchy or any indication of which processes represent a logical business process. Collective persistence behaviour on certain higher levels is therefore not directly supported by the model. However, it can be easily implemented based on the primitives provided. To implement such collective behaviour and to better separate the persistence related logic it is possible that one process can persist another one provided that the latter is idle. This

allows defining custom persistence control processes, which can observe others and trigger persistence for a collection of processes based on certain criteria. For example, assume a logical business process including three concurrent control flows each implemented by one state-machine process. Furthermore, assume that the desired persistence behaviour would be to store all three processes persistently if all of them have been idle for more than two minutes. In this case, a fourth process could be used to monitor the three others, maintain a timer, and trigger the system to store the processes persistently if the timer expires. Monitoring may be implemented in different ways by, for instance, orchestrating the three processes to signal activity via messages to the monitoring process or by using the monitor as a message mediator.

A persistence provider is an external service that can be attached to the process engine. It must implement a well-defined interface through which the engine can store or retrieve process instance information under a given ID, typically the process handle. The structure of the information can be opaque to the persistence provider. Each persistence provider must be uniquely addressable such that processes can define the target provider when requesting persistence for themselves or for other processes.

Storing process instances persistently not only makes it possible to minimize resource consumption but also sets the ground for process durability. However, process durability in addition requires that the state of the process engine itself is stored persistently, too, which typically includes:

- Deployed process definitions;
- information on registered persistence providers and other external entities; and
- the persistence table and other relevant run-time structures such as data shared among process instances or the list of active (non-idle) processes.

Continuously maintaining this state in persistent memory in a transactional manner while the process engine is running would have negative impact on the execution performance. Thus, the engine must be stopped in a controlled way before it can be stored persistently and removed from system memory. This includes discontinuing the delivery of any pending messages, waiting for all process instances to finish their current message processing, and storing all process instances persistently. At this point, there might be process instances, which are active because they have pending messages. These messages are part of the persistent process instance state, yet the process engine itself still must additionally remember these processes in its own persistent state to be able to restart

them when the engine is restored later. In addition, the persistence providers must be flushed such that all data is written to the durable medium before the engine ceases to exist. Finally, the process engine terminates and returns its own state to the host application, which stores it persistently.

Note that whenever the engine is restored and restarted, it needs to verify that all persistence providers are re-attached before it continues normal execution. Only then, it restores all active processes from the persistence providers and continues delivering pending messages.

3.7.3. Persistence in ePVM

The persistence model introduced in Section 3.7.2 has been prototypically implemented in ePVM. In ePVM the mechanism for the runtime engine to initiate communication with the host application is via registered host process objects as discussed in Section 3.5.2. Thus, in ePVM a persistence provider simply is a normal host process object except that it must be able to process a certain set of persistence service messages. As all messages in ePVM, these messages consist of a JavaScript object with certain properties. In a first implementation, the three messages described in Listing 12 have been used. The messages are defined using the JavaScript object notation (JSON) [106] extended with JavaDoc comments. Based on this interface, a persistence provider has been implemented, which stores the process states in files using the process handles as file names.

```

/** persist process instance
 * @param hnd handle of the process to be persisted
 * @param s serialized process state
 * @return true if successful, false otherwise
 */
{id:"persist", handle:hnd, state:s}

/** retrieve process instance
 * @param hnd handle of the process to be retrieved
 * @return serialized process state
 */
{id:"retrieve", handle:hnd}

/** flush persistence provider */
{id:"flush"}

```

Listing 12. Persistence service messages (JSON with JavaDoc)

As mentioned in Section 3.7.2, the information to be stored persistently for a process instance includes its process state and associated thread. The corresponding items in ePVM are the so-called process environment including any state variables, the communication queues (in-queue and save-queue, respectively), and some supplementary

information about the process and its thread such as the process type and the thread's priority. To serialize this information, the ePVM engine serializes the relevant JavaScript objects. Here, the restriction is that the process state must not include references to objects shared with other processes. References to system specific objects such as the top-level scope or closures are replaced by stubs, which are resolved when the process instance is reconstructed.

An important point when implementing the proposed persistence model in ePVM is that the communication with a persistence provider must be synchronized and asynchronous to the process engine thread. The latter is important because the ePVM engine itself is single threaded, that is, waiting for a persistence provider to persistently store and retrieve a process state would block all other processes. Since a dedicated thread always delivers messages for host processes, these operations can be easily decoupled from the engine thread by sending messages asynchronously. However, it is equally important to synchronize the messages such that the host process receives them sequentially and in order. For instance, a request to store a process instance persistently must be processed before the same instance can be retrieved. To handle these issues for all process instances, a special system process, the so-called persistence device handler (PDH), has been added to the ePVM engine as illustrated in Figure 26. Whenever a process instance is to be stored persistently, the engine serializes its state and forwards it to the PDH, which takes care of asynchronous communication with the persistence provider and any store/restore synchronization. The process engine then simply marks the process instance as stored persistently. When the first message for a persistently stored process arrives, the PDH is triggered to retrieve and reconstruct the process instance. In the meantime, an in-queue is created and all messages sent to the process instance are queued in its in-queue but the process engine does not schedule the process until it has been fully reconstructed by the PDH.

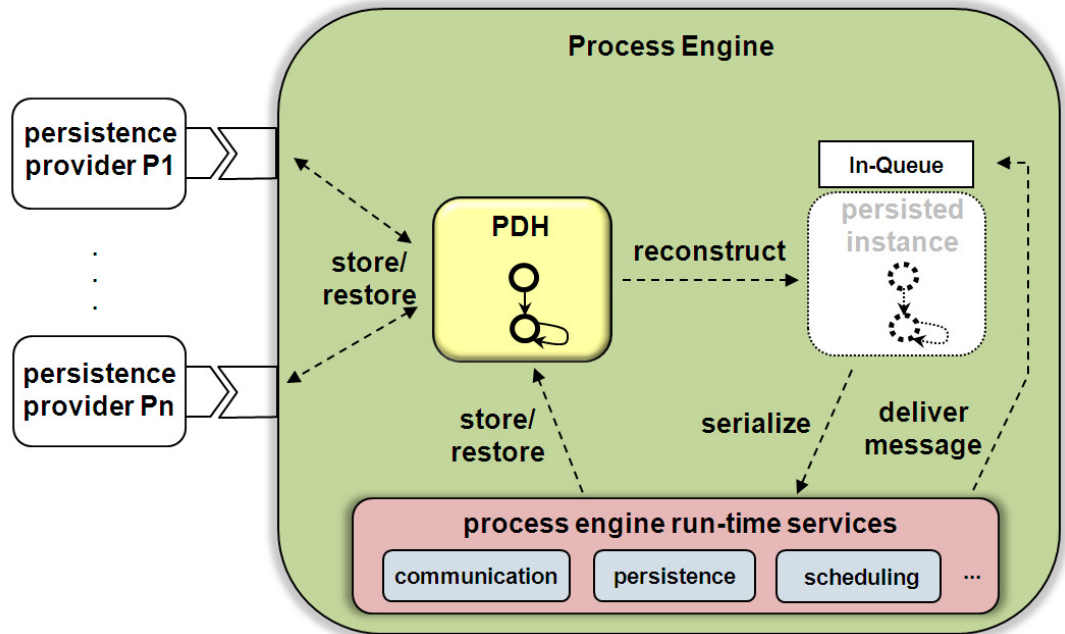


Figure 26. Asynchronous persistence operations via PDH

The JavaScript runtime API provided to processes for controlling their persistence behaviour is very easy. It solely consists of the function *PVM_persist()* taking the handle of the persistence provider as a parameter and optionally the handle of the process instance to be stored persistently. If the latter is not provided, it defaults to the process instance invoking the function. The code snippet shown in Listing 13 provides a simple example process, which requests persistence with different persistence providers depending on its state. In the state *initial contact* it requests persistence via the persistence provider named *initial customer contact DB* and in the state *customer acquired* the provider *customer DB* is used. The example illustrates how easy it is to control persistence on the level of individual messages.

The process engine itself is stored persistently by serializing the complete run-time state of the JavaScript interpreter. This is possible because the interpreter supports capturing and serializing continuations. The ePVM engine itself is mostly implemented in JavaScript such that the continuation automatically includes all relevant run-time structures such as the registered host processes or the persistence table.

```
function persistence__PVMProcess(environment, message) {
    var provider;

    if (environment.state == null) { // initial message
        // retrieve persistence provider handle
        provider = PVM_hostProcess("initial customer contact DB");
        environment.state = "initial contact";
    }
}
```

```

        PVM_persist(provider); // request persistence
        return true;
    }
    else if (environment.state == "initial contact") {
        provider = PVM_hostProcess("customer DB");
        environment.state = "customer acquired";
        PVM_persist(provider); // request persistence
        return true;
    }
    else if (...)
        ...
}

```

Listing 13. Flexible persistence example (snippet)

3.8. THE ePVM PROTOTYPE IMPLEMENTATION

3.8.1. Overview

Within the context of this thesis, a prototype implementation of the ePVM framework including the features described in this chapter has been developed. The implementation is based on the Mozilla Rhino interpreter [30] representing the basic JavaScript execution environment. Rhino is an open source JavaScript interpreter written in Java. The ePVM prototype embeds the Rhino engine and it is available as a Java library, too. Rhino is the only third-party package that needs to be available for ePVM to be operational. Apart from this dependency, it is fully self-contained, which is an important feature enabling ePVM to be easily embedded into all kinds of application scenarios. For the same reason, it does only require a standard Java runtime environment as opposed to a server-based J2EE environment, which would limit the field of application.

Another important design decision is the fact that the ePVM runtime system itself is almost completely implemented in JavaScript. Only the host API and a few basic helper functions, so-called *native functions*, have been implemented in Java as illustrated in Figure 27. The host API obviously needs to be implemented in Java as ePVM is available as a Java library. Nevertheless, attention was paid to keep this API layer as thin as possible. The same applies for the native functions. Only functionality that could not be implemented in JavaScript, for example, the management of OS threads, has been made available as a native function. The bridge between the JavaScript part of ePVM and the native functions implemented in Java is the so-called *LiveConnect* feature provided by Rhino. Essentially, it enables the invocation of Java methods from JavaScript code and vice versa (c.f. Figure 27).

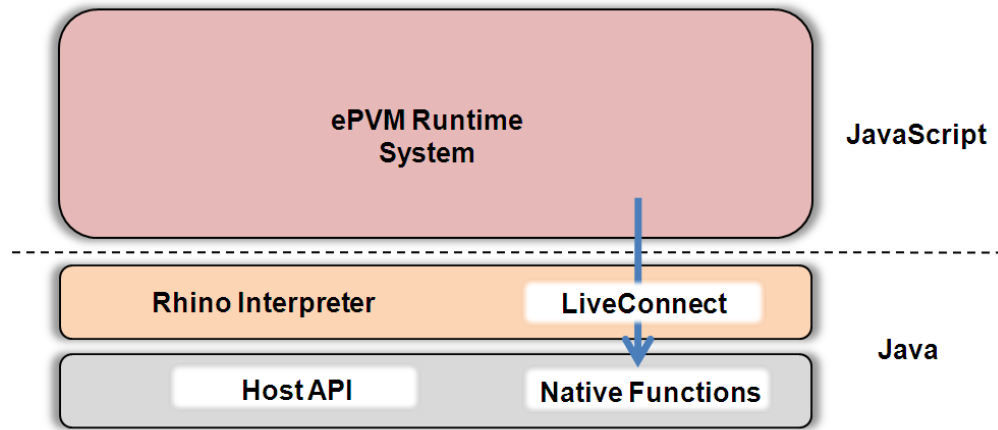


Figure 27. The ePVM prototype implementation

The fact that ePVM is mostly written in JavaScript keeps it independent from the underlying interpreter and the language mapping of the ePVM library itself. As a result, the effort of moving, for instance, from a Java to a C library by using a C version of the JavaScript interpreter and implementing the host API and native functions in C becomes manageable. In other words, it does not require a complete rewrite of the runtime system in C. In order to hide the source code of ePVM, the JavaScript source is compiled into Rhino specific interpreter byte code. The ePVM sources are then packaged into the library as a resource in this byte code format.

3.8.2. Host Application Programming Interface

This section provides a brief overview of the most relevant host API methods available via the Java class *ProcessVM*. It gives an idea of the functionality that is available for embedding ePVM into other applications, deploying process definitions, working with process instances, and communicating with process instances. Additionally, it eases understanding the source code listings provided throughout this document. In general, this API is available to the host application as well as to host process objects. However, some of the communication functions can only be used by host process objects invoked by the ePVM runtime. For example, the *reply()* function obviously only works if called by a host process object, which has a current message to reply to. Furthermore, it is to be noted that the host application as well as host process objects cannot communicate with process instances of type *HANDLER*, as they do not run in an ePVM thread, which the handler process could borrow for execution. Table 1 summarizes the most relevant API methods (see also Appendix A.1).

Table 1. ePVM host API methods

Basic Methods	
ProcessVM()	Constructor creating an instance of the ePVM process execution engine.
deploy()	Deploys process definitions in form of a process package to the engine.
start()	Start the execution engine. This means that the engine starts executing (scheduling) process instances.
stop()	Stops the execution engine.
buildObject(data)	Creates a <i>PVMObject</i> representing a JavaScript <i>Object</i> , <i>Array</i> , or <i>XML</i> object. Such objects can be used in communication-related methods to exchange complex data between the host application/process and ePVM processes. Optionally, a JSON string or an XML string can be provided via the data parameter to further define the object to be built. If the data parameter is <i>null</i> , an "empty" JavaScript <i>Object</i> object is generated. An XML string generates an XML object as defined in the E4X (ECMAScript for XML) specification [104]. If an array object is passed to the method, e.g. <i>int[]</i> , a corresponding JavaScript <i>Array</i> object is generated.
invoke()	Invokes a process by creating a new process instance.
join()	Creates a process instance that joins a given process instance to form a process group (sharing one ePVM thread).
registerHostProcess()	Register a host process object with the ePVM engine under a unique name. Registering host process objects makes them visible for process instances running within the engine.
registerMonitor()	Registers a process monitor for a given process package with the ePVM runtime. The monitor receives all events indicated in the event mask passed to this method. The ePVM runtime can deliver monitor events either asynchronously or synchronously. In the latter case, the runtime guarantees that the process which has triggered the event does not continue its message processing until the monitor has processed the event. A process of type

	<i>HANDLER</i> cannot be monitored synchronously, since it must not be blocking.
info()	Provides information (e.g. type, name) about a given process.
packageList()	Lists currently deployed process packages.
processList()	Lists all process instances of a given process package, or all registered host process objects.
Communication-related Methods	
call()	Sends a message to a process instance synchronously.
send()	Sends a message to a process instance asynchronously.
reply()	This method can be used by host process objects to send a reply to the current message.
forward()	This method can be used by a host process object to forward the current message. The message value might be modified or not.

3.8.3. Runtime Application Programming Interface

This section provides a brief overview of the most relevant JavaScript runtime API functions available for the implementation of process definitions. It gives an idea of the functionality that is available for working with process instances, managing the save-queue, and communicating with process instances. Additionally, it eases understanding the source code listings provided throughout this document. By design, the runtime API is very similar to the host API. In particular, the functions for process instantiation and communication are intentionally kept similar with the goal to provide a very simple and intuitive API. The major difference of the runtime API is that it additionally provides save-queue related functions. Table 2 summarizes the most relevant runtime API functions (see also Appendix A.2).

Table 2. ePVM runtime API functions

Basic Functions	
PVM_invoke ()	Invokes a process by creating a process instance.
PVM_join()	Creates a process instance that joins a given process instance to form a process group.
PVM_hostProcess()	Retrieves the handle of the host process object with a given name.

PVM_info()	Provides information (e.g. type, name) about a given process.
PVM_monitor()	Sends a custom monitor event to the process monitor that is monitoring the process package to which the calling process instance belongs.
PVM_self()	Retrieves the handle of the process instance calling this function.
PVM_yield()	Temporarily pauses the process and yields control. Other processes with equal or higher process priority might be scheduled.
Save-queue-related functions	
PVM_save()	Moves the current message, if any, of the currently executing process instance to its save-queue. The current message will be <i>null</i> afterwards.
PVM_retrieve()	Retrieves data from the save-queue. Value or sender of a particular message in the save-queue of the currently executing process instance can be retrieved. Optionally, the message can be removed from the queue.
PVM_purge()	Discards all messages from the save-queue of the currently executing process instance.
PVM_sort()	Defines a sort function and sorts the save-queue. By default the save-queue is organized LIFO. However, this function allows defining a custom compare function for the queue. The queue is immediately sorted using the given compare function. Furthermore, the compare function is used to enqueue any subsequently saved messages.
Communication-related functions	
PVM_call()	Sends a message to another process instance or host process object synchronously.
PVM_send()	Sends a message to another process instance or host process object asynchronously.
PVM_forward()	Forwards the current message to another process instance or host process object.
PVM_forwardSaved()	Forwards a message from the save-queue to another process instance or host process object.

PVM_receive()	Explicitly receives a new message from the in-queue. The new message, if any, becomes the current message of the process. This function might be blocking or non-blocking depending on the mode parameter.
PVM_reply()	Sends a reply to the current message.
PVM_replySaved()	Sends a reply to a message in the save-queue. The message will be removed from the save-queue.
PVM_sender()	Returns the handle of the sender process of the current message.

3.8.4. The Timer Process

The current ePVM prototype implementation includes a built-in process providing timer services to other processes. Built-in means that this process is by default always available while the ePVM engine is running. Timers are frequently required within process definitions to trigger actions upon some delay or at a specific point in time. Thus, processes can set up timers using absolute and relative times by providing a date or duration. A process can set up an arbitrary number of timers by sending timer messages to the timer process including a timer ID as defined in Appendix A.2. The timer process has the process handle *PVM_TIMER*. Timer messages might be sent synchronously or asynchronously. If a timer expires, the original timer request message, possibly including some supplementary information such as the expiration date, is bounced back to the requesting process instance.

Internally, the timer process is implemented via a standard ePVM process of type *HANDLER*. The only difference is that the scheduler of the ePVM runtime system regularly triggers the timer process between context switches to ensure timely delivery of timer messages.

3.8.5. The Stock Quote Example Application

This section provides a more elaborate example of a process-driven application, the stock quote application, with the goal to illustrate the power of the ePVM framework in a real-world scenario. The purpose of the application is to monitor company stock prices and raise an alert if a stock is moving more than X dollar in a given timeframe T. An arbitrary number of different stocks can be monitored whereas for each stock X and T can be specified individually. Stock price information is retrieved from an available financial information Web service. Furthermore, a graphical user interface (GUI) lists the

monitored stocks along with their current price and an alert GUI pops up if an alert is detected.

The main control logic required for monitoring a stock is implemented as an ePVM process. For this purpose, the ePVM engine is embedded into a Java host application as illustrated in Figure 28. The process instances make use of two host process objects, one for accessing the Web service providing stock prices, and one for triggering an alert by popping up a simple graphical alert notification dialog window. Furthermore, a process monitor object is used to trace the state of the process instances and visualize it in a simple text window, the so-called monitor GUI as shown in Figure 28. As the focus lies on the ePVM framework, the user interface is very basic. Users can start and stop processes via a command line interface provided by the host application. To be able to retrieve stock prices on a regular basis, the process instances take advantage of the built-in timer process provided by ePVM (c.f. Section 3.8.4).

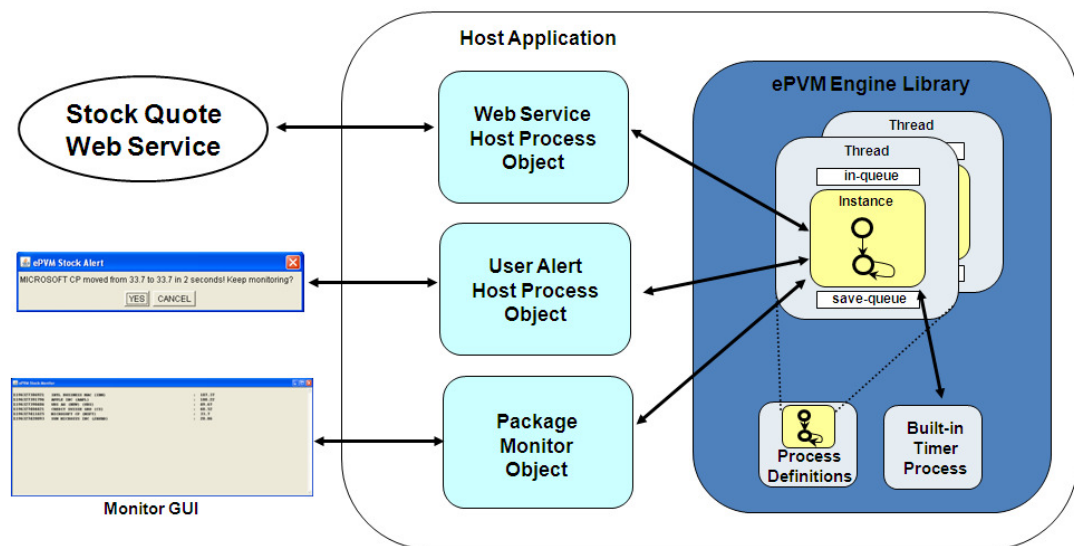


Figure 28. **Stock quote application architecture**

The idea is to define the process required to monitor one stock as an ePVM process and then create an arbitrary number of instances of this process, one for each stock to be monitored. This represents a typical business process scenario in which using a process engine for handling the high level of concurrency makes a lot of sense. Figure 29 provides a flow chart describing the functionality required for monitoring a stock. Initially, the process gets started with three parameters defining the stock symbol of the stock to be monitored, the change in dollars that triggers an alert, and the timeframe in which the change must happen to trigger the alert. The core monitoring loop consists of

retrieving the stock quote from the Web service, updating the monitor GUI, and then waiting for the given timeframe before retrieving the stock quote again. When the stock quote has been retrieved, it must be checked if an alert needs to be triggered or not. In the earlier case, the alert GUI must be popped up and the user can indicate whether to keep monitoring the stock or not. Figure 29 indicates which activities in the process flow trigger external entities. The *get quote* activity triggers the stock quote Web service via the corresponding host process object, the *update monitor* activity triggers the monitor GUI, and the *alert user* activity triggers the alert GUI.

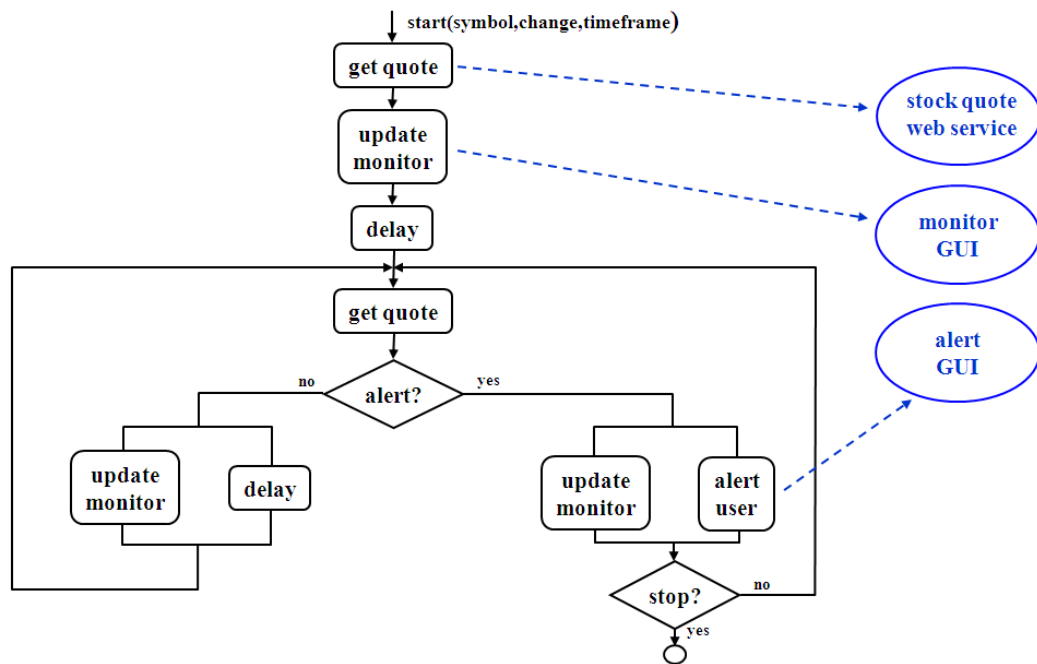


Figure 29. The stock quote process flow

Listing 14 exemplifies how the process flow for monitoring a stock can be implemented within the ePVM programming model. The function *StockMonitor* implements the main state machine that handles three different messages: the start message, the message indicating that a timer expired or an alert dialog has finished, and the stop message. For raising alerts, a separate process definition, the *raiseAlert* process, is used. Alert notifications as well as timers, implemented via the built-in timer process, run in parallel to the processing of monitor events. This together with the fact that a large number of *StockMonitor* process instances may exist, leads to a very high degree of concurrency within the application. The example illustrates how easy it is to cope with concurrency and synchronization in the ePVM programming model by combining the concept of CEFSM with the simplicity of JavaScript.

```

__PVMPackage = "Stock Quote v0.1";

// Public ePVM process to monitor stocks.
//
// start message:
// {id:"start", symbol:<1>, change:<2>, timeframe:<3>}
//   <1> stock symbol
//   <2> quote change per timeframe to trigger an alert
//   <3> timeframe in seconds
//
// timer expired/alert finished:
// {id:1}
//
// stop message:
// {id:"stop"}
//   terminate monitor process
function StockMonitor__PVMPProcess(env, msg) {
    var quote, alert;

    switch(msg.id) {
    case "start":
        env.target = msg; // remember start message
        // and host process handle
        env.sqHnd = PVM_hostProcess("Stock Quote Web Service");
        // call web service (synchronously)
        quote = PVM_call(env.sqHnd, msg.symbol);

        env.last = Number(quote.Last); // remember last quote
        env.company = String(quote.Name); // and full company name

        // send stock info to monitor (monitor is synchronous)
        PVM_monitor({company:env.company, symbol:msg.symbol,
            last:env.last, alert:false});

        // set up timer asynchronously
        PVM_send(PVM_Timer, {mode:"duration",
            msec:(msg.timeframe * 1000), id:1});
        PVM_reply("ok"); // send reply to start message
        break;

    case 1: // timer expired/alert finished
        // call web service synchronously
        quote = PVM_call(env.sqHnd, env.target.symbol);

        if (alert = (Math.abs(env.last-quote.Last) > env.target.change))
            // create alert process and send info asynchronously
            PVM_send(PVM_invoke(raiseAlert), [env.company, env.last,
                Number(quote.Last), env.target.timeframe]);
        else // keep going, set up timer
            PVM_send(PVM_Timer, {mode:"duration",
                msec:(env.target.timeframe * 1000), id:1});

        PVM_monitor({last:Number(quote.Last), alert:alert});
        env.last = Number(quote.Last); // remember current stock price
        break;

    case "stop":
        PVM_send(PVM_Timer, {mode:"stop"}); // stop all timers
        return false; // terminate process
    }
}

```

```

    return true; // process has not terminated
}

// Package internal ePVM process to trigger stock alert
//
// alert message:
// [<company-name>, <last-quote>, <current-quote>, <timeframe>]
function raiseAlert(env, msg) {
    var res;

    // send message to alert host process synchronously
    res = PVM_call(PVM_hostProcess("Stock Quote Alert"), msg);

    if (res == "stop") // user wants to stop monitoring this stock
        PVM_reply({id:"stop"}); // reply with stop message
    else
        PVM_reply({id:1}); // reply with alert finished message

    return false; // terminate process
}

```

Listing 14. Stock quote process definition

3.9. SUMMARY AND CONTRIBUTIONS

The ePVM framework, as introduced in this chapter, represents a generic core framework for process execution. It addresses the requirements and challenges discussed in Section 1.8.1 by providing:

- a lightweight, generic, and easily embeddable process execution engine that can be used to build process-driven applications as well as comprehensive business process management solutions supporting multiple domain specific process languages;
- a process model based on the theoretical foundation of CEFSM enabling efficient modelling of event-driven concurrent process flows;
- a programming model that combines the concept of CEFSM with the simplicity of JavaScript leading to an easily programmable process execution environment;
- a flexible and dynamic persistence model that enables fine grained persistence control and thus leads to optimized performance and resource consumption;
- a prototype implementation along with sample applications as a proof-of-concept and as the basis for evaluation.

We believe that this unique combination of features incorporated in the ePVM architecture will prove superior to today's domain specific, monolithic process execution environments lacking interoperability, versatility, and programmability. This chapter is based on work published in [48,49].

4. SECURE MULTI-PARTY TRANSACTION

AUTHORIZATION

Having identified the challenges and requirements of secure remote authentication and transaction authorization (c.f. Sections 1.3 and 1.8.2), and having discussed the limitations of existing approaches, a novel approach providing not only the highest level of security, but also addressing other aspects such as ease-of-use, mobility, integration, and administration is presented in this chapter: the *Zone Trusted Information Channel* (ZTIC). After introducing the basic ideas behind the ZTIC, the two main generic usage scenarios are presented. Afterwards, it is described in more detail how the ZTIC approach addresses the given challenges and requirements followed by a discussion on remaining attacks and possible remedies. Finally, it is described how the concept of the ZTIC has been integrated with the ePVM framework to enable highly secure multi-party transaction authorization.

4.1. THE ZONE TRUSTED INFORMATION CHANNEL

4.1.1. The Basic Approach

As discussed in Section 1.8.2, the core property of any new approach certainly is resistance not only against phishing attacks, but also against MSW and MITM attacks. Given today's hardware and software environment provided by a typical end-user PC, it is well known that such a high level of security can only be reached via the use of an additional trusted device – the so-called security token – which is equipped with a display and some sort of keypad. Solutions involving, for example, Secoder [28] compliant smart card reader devices or EMV CAP based tokens (c.f. Section 2.2.6) illustrate this. All these devices are used, at some point within an online session when authentication or transaction authorization is required, for digitally signing some data in order to proof that the user is authentic and the data is correct. In this chapter, a new approach is presented, which somehow has got the name Zone Trusted Information Channel (ZTIC, pronounced 'stick'). The ZTIC approach is also based on a security token that includes display and buttons, however, it is different from existing approaches in many other respects discussed throughout this chapter. The fundamental technical properties that make the approach unique are:

4. Secure Multi-Party Transaction Authorization The Zone Trusted Information Channel

1. The ZTIC adds a trusted and tamper-resistant secure communication endpoint to the otherwise untrustworthy client PC. Through this endpoint, a user can securely communicate with sensitive remote services. In other words, the ZTIC security token establishes a secure end-to-end communication channel between itself and the remote service. Additionally, it adds a secure display, buttons, and smart card reader to the communication endpoint.
2. The ZTIC is solely using prevalent standard protocols and interfaces such as SSL/TLS, HTTP, or USB.

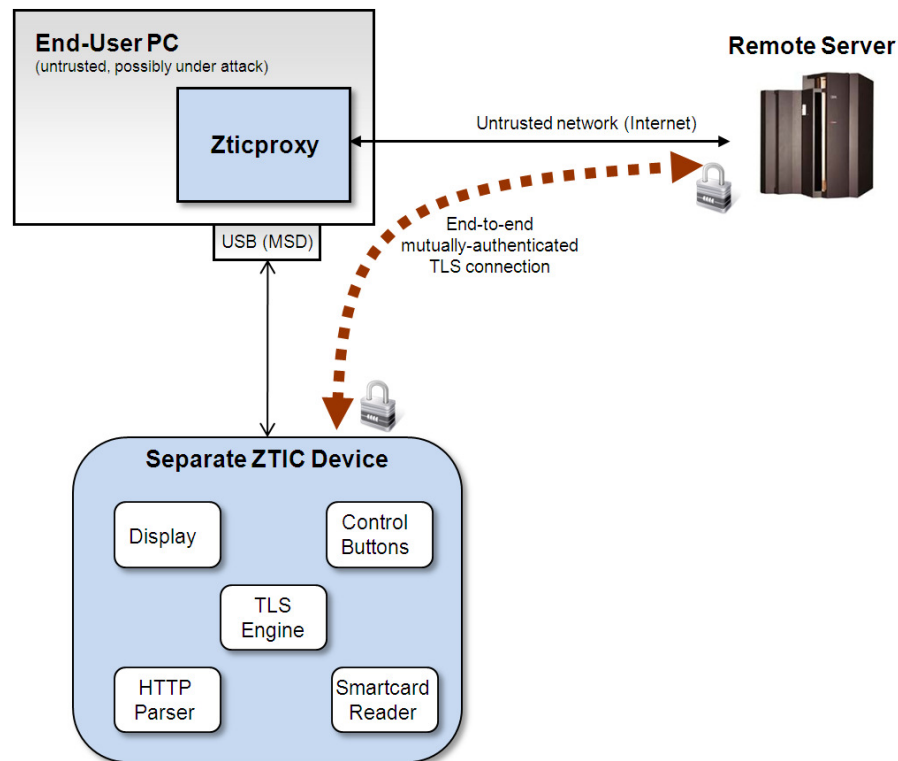


Figure 30. Basic ZTIC configuration

Figure 30 shows the basic configuration of the ZTIC device and illustrates the usage scenario. Conceptually, the ZTIC hardware consists at minimum of a processing unit, volatile and persistent memory, a small display, at least two control buttons, and optionally, a smart card reader. The two buttons are typically used as an *OK* and a *Cancel* for the user to approve or decline authentication or transaction operations. Its software is minimally configured with a complete TLS engine including all cryptographic algorithms required by today's SSL/TLS servers, an HTTP parser for analyzing HTTP traffic exchanged with the server, plus a custom system software implementing the USB mass storage device (MSD) profile. The latter is required to connect the ZTIC device to the

4. Secure Multi-Party Transaction Authorization The Zone Trusted Information Channel

end-user PC as illustrated in Figure 30. The USB MSD profile has been identified as the interface that is most likely to be available in today's standard PC hardware. This approach makes the ZTIC dependent only on standard USB MSD device drivers readily available on all operating systems. Other interface types such as smart card reader or network device have been considered as well. However, all of them required a custom device driver on some or all operating systems. The communication via USB MSD is realized by writing and reading a dedicated file available in the ZTIC file system. This file is not a real file in the ZTIC's persistent memory but only a virtual file mapped to internal transient memory. The *zticproxy* alternately writes requests into this file and then reads corresponding results from it. All other files in the ZTIC file system, for instance, the *zticproxy* executable are read-only. In other words, there is no way to write persistent memory via the USB interface.

The main reason why the ZTIC needs to be connected to the end-user PC is to get network access. The goal is to establish an end-to-end mutually authenticated TLS connection between the ZTIC and the remote server. Obviously, for that the ZTIC needs network access. It has been decided that the ZTIC makes use of the network connection of the end-user's PC via the help of a networking proxy – the so-called *zticproxy*. Alternatively, the ZTIC could have its own network interface, for example, via a built-in WLAN or UMTS module. However, this would have significantly increased hardware and software complexity and thus eventually the cost of the ZTIC device, which ultimately influences the success of the overall approach. The *zticproxy* solely forwards network packets between the ZTIC and the remote sever, yet, since running on the potentially insecure end-user PC, itself does not contribute to the ZTIC's security. In other words, the ZTIC does not rely on the *zticproxy* to behave properly; in contrast, it is at any point in time considered potentially malicious. Thanks to the use of the USB MSD interface, the *zticproxy* can be pre-loaded on the ZTIC and started straight from there. Credentials required for establishing the mutually authenticated TLS connection are either securely stored in the ZTIC's persistent memory or, even more tamper resistant, on a smart card that has to be inserted into the ZTIC's smart card reader. Such credentials typically comprise one or more server certificates used in TLS server authentication and some secret or private key used in TLS or application-level client authentication.

In the given configuration, the ZTIC can display either information sent by the remote server or information to be sent to the remote server. In both cases, the ZTIC displays such information on its built-in display and requests the user to verify the information

4. Secure Multi-Party Transaction Authorization The Zone Trusted Information Channel

and either approve or decline by pressing the appropriate button on the ZTIC. This way, the user has the opportunity to check and approve data on a trusted platform as opposed to doing this on an untrusted PC display and keyboard. Data exchanged between server and ZTIC is formatted according to the HTTP protocol, such that the ZTIC is compatible with all existing standard Web servers, as all of them support TLS as well as HTTP. In principal, it is not necessary that transactions authorized by the user be digitally signed by the ZTIC because confidentiality and originality are implicitly guaranteed by the TLS channel. However, the ZTIC can optionally do this via a smart card that can generate digital signatures to implement legally binding non-repudiation.

4.1.2. In-Stream Usage

The basic ZTIC configuration as described in Section 4.1.1 can be utilised in real-world authentication and transaction authorization scenarios in two different ways – the so-called *in-stream* usage scenario and the *second channel* usage scenario. This section describes the in-stream usage scenario, which is particularly useful in applications such as online banking where typically single-party authentication and transaction authorization operations are carried out synchronously. In such applications, the end-user starts a client application on his or her PC, which establishes a connection to the remote server, for example, a remote workflow server, using an untrusted network such as the Internet. At the time of writing, applications tend to be more and more browser-based such that the client application oftentimes actually is a Web browser. Figure 31 illustrates how the ZTIC can be integrated into such an application scenario. In-stream integration means that all communication between the client application and the remote server is passed through and processed by the ZTIC, which in turn is hooked into the communication path via the *zticproxy* running on the PC. When in use, the ZTIC continuously scans the data exchanged between client application and server for sensitive operations. In the context of online banking, for example, a sensitive operation would be a money transfer. For each sensitive operation, it intercepts the communication flow, extracts crucial information for display and verification, and proceeds only after the user has explicitly confirmed the operation by pressing the ok button on the ZTIC device. Non-sensitive operations are just passed along without user interaction.

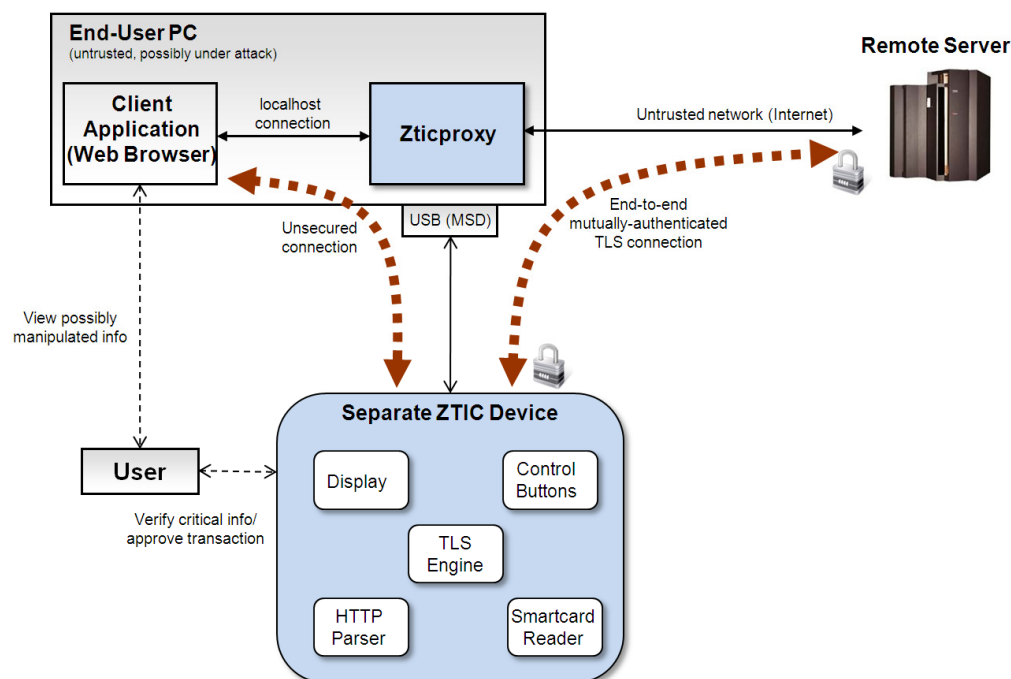


Figure 31. ZTIC in-stream usage scenario

Referring to Figure 31, a typical client/server interaction involving the ZTIC follows these steps:

1. The user plugs the ZTIC into his or her (potentially insecure) PC, which mounts the ZTIC as a USB mass storage device and auto-starts the zticproxy application also residing on the ZTIC. In cases where the operating system does not support auto-start of applications from removable devices, the zticproxy must be started manually.
2. The proxy in response triggers the ZTIC to initiate and run a TLS session handshake including mutual authentication. Hereby, the zticproxy “blindly” relays all the communication exchanged between ZTIC and server, that is, without being able to interpret the communication itself. Upon successful completion of the handshake, both client and server have been authenticated and a TLS session has been established between ZTIC and the remote server.
3. From then on, all interactions between the client application and the server, still “blindly” relayed by the zticproxy, are scanned by the ZTIC’s built-in HTTP parser for sensitive operations. If found, the ZTIC sends the user’s operation request and transaction data only after explicit user consent has been given as described before. For instance, considering again a money transfer in the context of an online banking application, the ZTIC extracts the transactions crucial data

4. Secure Multi-Party Transaction Authorization The Zone Trusted Information Channel

such as the recipient's bank account information and the amount of money to transfer, displays this information on its display, and awaits the user's explicit confirmation via a press on its ok button before proceeding. If the user detects any manipulations in the transaction's crucial data, he or she can abort the transaction by pressing the cancel button.

4. Likewise, confirmation data from the server can also be dynamically extracted and shown on the ZTIC's display for user verification, if needed.

So, how does this scheme defend against MITM and MSW attacks? Fundamentally, it is impossible to prevent MITM/MSW attacks on insecure client PCs. Even with a TLS connection between the client application and the server or the zticproxy, respectively, an attacker can still get hold of and modify the plain data, for instance, by tampering with the client application. The so-called man-in-the-browser attack represents a well-known example. The ZTIC therefore does not try to prevent MITM/MSW attacks to begin with - in fact, it cannot - but instead focuses on exposing MITM/MSW attacks to the user who then can take appropriate action. Being a MITM itself, the ZTIC makes explicit what really is communicated between client and server, and while a MITM/MSW may still passively read the transaction data exchanged, any modification will be inevitably disclosed by the ZTIC. Only sloppy verification of the transaction data shown on the ZTIC display by the user will allow modified transaction data still to be sent to the server.

Furthermore, provided the user credentials and the certificates for TLS mutual authentication are maintained securely on the ZTIC, for example on an embedded smart card, no MITM/MSW can impersonate the ZTIC while the ZTIC in turn can verify that it connects to a genuine server. Since the zticproxy only implements a relay service, any attack on the proxy may, at worst, just cease the service to work with no other harm done. Note that while the proxy may communicate with the ZTIC in plain, all the (user-confirmed) communication between the ZTIC and the server is protected by the TLS session terminating in the ZTIC, thus keeping MITM/MSW locked out from modifying transaction data unnoticed.

It should be noted, though, that for the security of this approach it is crucial to keep the HTTP parsers evaluating all transaction data operating exactly the same on the ZTIC as on the server. Otherwise, the ZTIC could be tricked to show transaction information different to the one processed by the server later on. In real-world deployments, this can

4. Secure Multi-Party Transaction Authorization The Zone Trusted Information Channel

represent a challenge as the server-side HTTP parsing code might be hidden deep in the Web server framework. Determining its exact behaviour might be very hard.

4.1.3. Second Channel Usage

While in the in-stream usage scenario all communication traffic between the client application and the server is routed through the ZTIC, in the second channel usage scenario two independent communication channels are used, the information channel and the security channel. The basic setting is similar to the one used in the mTAN solution discussed in Section 2.2.5, security properties are different though. Figure 32 illustrates the approach. The client application communicates directly with the remote server, just as if the ZTIC would not be used at all. This connection, which is referred to as the information channel, might or might not be secured via SSL/TLS. In any case, it is potentially subject to MITM or MSW attacks. However, for sensitive operations such as authentication and transaction authorization a second channel, the security channel, is used. This security channel is provided by the ZTIC as shown in Figure 32. On the client side, the two channels are technically completely independent, only the user links them together. As a result, the ZTIC might either be used on the same PC as the client application or on any other network connect PC.

Referring to Figure 32, a typical client/server interaction involving the ZTIC in second-channel mode follows these steps:

1. The user starts the client application, which in turn establishes the information channel by connecting to the remote server. In the login phase, the user indicates to the server who he or she claims to be.
2. The server checks if it already has a second connection, the security channel, with the ZTIC device that belongs to that user. If not, the server requests the user, via the information channel, to activate his or her ZTIC.
3. The user activates his or her ZTIC device by connecting it to a PC via USB and starting the zticproxy as described in Section 4.1.2.
4. When the zticproxy is started, it immediately initiates the establishment of the TLS session (c.f. Section 4.1.2) between the ZTIC and the server and upon that sends a HTTP request to the server via the ZTIC.
5. As soon as the server receives the request, it checks if it has any pending operations for this ZTIC. In this case, it would have a pending authentication request, since the user is currently about to log in. The server thus sends a

4. Secure Multi-Party Transaction Authorization The Zone Trusted Information Channel

response to the ZTIC including, for instance, a random number. At the same time, it sends the random number to the client application via the information channel.

6. The ZTIC displays the random number on its internal display and requests the user to approve the authentication operation via the ok button on the ZTIC.
7. The user compares the random number shown on the ZTIC with the random number shown by the client application to ensure that the two belong together and then approves by pressing the ok button on the ZTIC. The user has now successfully logged in to the application and may view sensitive information via the information channel.
8. If at some later point in time the user submits a sensitive transaction request over the information channel, for instance, a money transfer, the server again displays this information on the corresponding ZTIC and awaits user approval via the security channel.

For the server to be able to initiate the verification of a transaction on the ZTIC at any point in time, the ZTIC continuously sends HTTP requests to the server (one after the other). The server does not answer such a request immediately if it does not have a pending operation for the ZTIC. In contrast, it keeps it as a pending request and only sends a response if a pending transaction is available or some timeout period expires. In the latter case, the server sends a timeout response message upon which the ZTIC simply submits a new HTTP request. This is a well-known technique known from so-called push-email services as, for instance, used by Blackberry. Obviously, the drawback is the fact that the server must be able to handle a large number of concurrent HTTP connections, or more precisely, HTTPS connections as TLS is used underneath. However, the load can be managed by applying certain ZTIC deactivation strategies depending on the application scenario. For instance, the ZTIC could be automatically deactivated after a certain idle time or as soon as the information channel is closed.

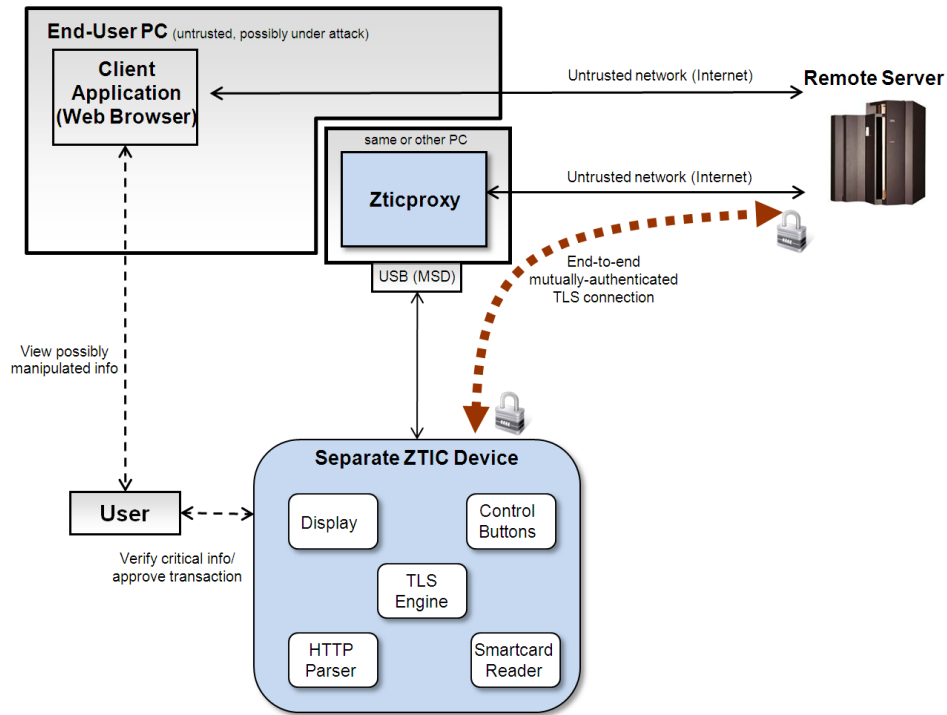


Figure 32. ZTIC second channel usage scenario

From a user perspective, both ZTIC usage scenarios, in-stream or second channel, can be very similar depending on the exact implementation. However, the security properties of the second channel usage scenario are slightly different compared to the in-stream scenario. In the latter case, classical MITM attacks, where an attacker is located somewhere in the network are impossible, as all communication is routed through the ZTIC. In the second channel scenario, in contrast, such an attack is still possible on the information channel, as it is not secured via the ZTIC. However, only eavesdropping can be achieved, manipulated transactions would still become visible as soon as they are communicated via the security channel. Furthermore, eavesdropping cannot be avoided as soon as a MSW attack is launched. A virus or Trojan horse residing on the end-user PC can always eavesdrop, for instance, the PC display or keyboard. Therefore, under the assumption that MSW attacks are possible and not significantly harder to launch than MITM attacks, and we believe that this assumption holds, both ZTIC usage scenarios can be considered equivalent from a security perspective.

4.1.4. Multi-Party Transaction Authorization

The example of online banking, as used multiple times throughout this chapter, exemplifies an application scenario where transaction authorization is typically synchronous, meaning that it is carried out at the point where transaction requests are

4. Secure Multi-Party Transaction Authorization The Zone Trusted Information Channel

submitted to the server from the client application, and carried out by a single person. However, in the business environment asynchronous and multi-party transaction authorization operations are required as well. Here, multiple people are involved when critical decisions are to be made during a business process flow. For example, a purchasing process might be triggered by an employee who is entering an initial purchasing request into an information system. At a later point within the process, one or more people of the employee's management chain might be required to authorize the purchase such that the necessary funds become available. In such cases, multi-party transaction authorization is frequently required for various reasons such as organizational hierarchies or risk management. Furthermore, if multiple people need to authorize a transaction, it is desirable that these authentication operations can be carried out asynchronously. This means, the transaction request is not immediately processed by the server, but it is just stored as a pending request for an arbitrary amount of time, independent of the connection to the client application that has submitted the request in the first place. Only if all required people have approved the transaction, the server executes it. Furthermore, it is desirable that a person that has to approve a transaction does not necessarily need to use the client application to do this. In other words, this person does not need to establish the information channel, but only the security channel as defined in Section 4.1.3. The second channel usage scenario of the ZTIC provides an ideal security framework to implement such asynchronous multi-party transaction support. Additionally, the use of the second channel approach offers the following general advantages over the in-stream approach:

1. There is no need to directly interface the client application with the ZTIC or zticproxy, respectively. This simplifies the approach.
2. The communication bandwidth for the information channel between the client application and the remote server is not limited by the ZTIC.
3. The ZTIC can be used independent of the client application. This is, on another PC or without the client application at all.

Section 4.4 exemplifies how the ZTIC can be embedded into a process driven server environment to implement asynchronous multi-party transaction authorization.

4.1.5. Ease-of-Use and Mobility

What makes the idea of the ZTIC unique is the fact that the approach does not only provide the highest level of security but also incorporates other very desirable properties,

4. Secure Multi-Party Transaction Authorization The Zone Trusted Information Channel

ease-of-use being one of those. In this respect, the ZTIC certainly represents the most advanced approach known at the time of writing. The user only needs to connect the ZTIC device, verify the information shown on its built-in display, and press one of the two buttons to approve or cancel an authentication or transaction operation. There is no need to input transaction details or transaction specific challenge values, nor is it required to manually copy short time codes from the device to the PC. At the same time, the ZTIC approach enables mobility, which additionally contributes to the ease-of-use experience from a user perspective. The simplicity of the ZTIC hardware design allows building ZTICs in very small and convenient form factors similar to key-fob-size USB memory sticks or MP3 player devices. Section 4.3 shows some of the ZTIC hardware prototypes. As a result, the usage pattern appears familiar and the device can be easily carried around, for instance, while travelling.

The design decision to attach the ZTIC to the PC using the USB MSB profile also significantly contributes to ease-of-use and mobility. Firstly, USB ports are nowadays available on virtually all PCs and, secondly, the user does not need to install any device driver software, since all important operating systems such as Microsoft Windows, Apple Mac OS, and Linux support USB MSD devices by default. Thus, there is a very high probability that the ZTIC works on an arbitrary PC, for instance, somewhere in an Internet café, just as it works at home.

4.1.6. Administration and Integration

For a solution to become widespread and thus commercially successful, its properties in the areas of administration and integration are critical. The latter targets the effort to be made to introduce a solution, which in most cases means integrating it into an existing environment. Administration rather targets ability to maintain a solution over time. Due to the direct TLS link between the server and the ZTIC device, it is very easy to securely update the ZTIC firmware or reconfigure it, for example, to react on new security requirements. As the `zticproxy` also resides on the ZTIC, it can be updated in the same way as the firmware. Whenever the ZTIC establishes the TLS connection to the server, it reports its version information. The server can then send an appropriate software update, which the ZTIC automatically installs before it proceeds with normal operation. Thus, the user does not need to be involved for downloading and installing updates. Only if the `zticproxy` is updated, the ZTIC needs to be restarted after the update. For this, the ZTIC automatically deregisters from the USB bus and registers again such that the MSD device

shortly disappears and appears again. The only thing the user has to do after that is restarting the zticproxy again.

With respect to integration, the ZTIC has a number of properties that make it very appealing for service providers. The most critical are:

1. Integrating the ZTIC into existing infrastructures requires minimal server changes. This is due to the fact that the ZTIC solely requires SSL/TLS and HTTP – protocols already available in almost every existing deployment. As a result, integration only involves standard Web server development skills. No specific server-side software components need to be developed or bought.
2. The ZTIC uses SSL/TLS for secure communication and server authentication. However, for client authentication, it can support other application level authentication mechanism as well. For example, it could use a EMV CAP compliant smart card or a list of pre-loaded one time codes to authenticate with the server. As a result, a service provider can reuse existing authentication protocols as well as existing client credentials. In other words, existing investments such as chip cards can be retained.
3. Due to ease-of-use, zero software installation, and compatibility with existing PCs, a service provider introducing the ZTIC can plan for minimal customer support requirements, a cost factor in real-world deployments that must not be underestimated.
4. The ZTIC hardware configuration enables the production of cost efficient devices. Other comparable solutions such as Secoder compliant smart card readers are, at best, in the same price range.
5. The support for secure remote software update and configuration retains investments, as the device can be dynamically adapted to new risks while in the field. For example, the ZTIC might be reconfigured to additionally require a ZTIC specific PIN code, or to implement a new cryptographic algorithm if the attack landscape changes over time.

4.2. ATTACKS AND REMEDIES

The ZTIC approach mainly focuses on software attacks such as MSW and MITM, as they are the ones that can be very easily launched in high volume. Physical attacks and social attacks are still conceivable. The latter are to some extent thwarted by the ZTIC because the user does no longer know the credentials (e.g. one-time codes) used and thus

cannot unveil them to an attacker. Therefore, the ZTIC is considered resistant against typical phishing attacks. However, the approach still relies on the human user to reliably verify the information showed on the ZTIC's display and only approve it if it is correct. An attacker may still launch social attacks to convince users to press the ok button even if the display shows undesired transaction information.

Besides social attacks, the following attacks are relevant and need to be considered in security evaluations:

1. **Stealing the ZTIC:** Stealing the device is a physical attack, which cannot be easily launched against a large number of users. Nevertheless, it is a feasible attack. To render this useless, the ZTIC can optionally be configured to be PIN protected. In this case, the user must enter a ZTIC specific PIN code, either directly on the device or via the client application, to unlock the ZTIC. A limited number of PIN entries further limits the risk. Consequently, an attacker would need to get hold of the PIN as well as the device itself. Furthermore, if a chip card is used that securely stores client credentials, the ZTIC itself does not need to be personalized with any secrets.
2. **Attacking the zticproxy:** This is possible as the zticproxy runs on the end-user PC and is thus subject to MSW. However, the zticproxy does not hold any secrets such that attacking it can "at best" lead to denial of service.
3. **Attacking the ZTIC:** The ZTIC might be attacked via software residing on the client PC or physically. The latter case only makes sense if the ZTIC stores secrets such as cryptographic keys. The level of security is then determined by the level of protection provided by the ZTIC hardware, for example, by tamper resistant microchips or sealed casing. Generally, it is recommended to rely on chip cards rather than storing secrets with the ZTIC's persistent memory.

With regards to software attacks launched from the client PC, the ZTIC does not provide a lot of contact surface. The only entry point is the virtual file used to communicate with the ZTIC. There is no other way to send any data to the ZTIC or access the ZTIC's persistent memory. Since this interface is kept very simple, the ZTIC firmware is able to reliably check for any anomaly such as buffer overflows or malformed requests. If an anomaly is detected, the ZTIC immediately stops operating and in certain situations, for instance, after a number of anomalies, it might also erase its own memory.

4.3. PROTOTYPES

The core ideas of the ZTIC and its various usage scenarios have been developed, patented, and published as part of this thesis work. However, a number of researchers have been working on implementing these ideas to produce prototype ZTIC hardware and software. This development work has not been part of this thesis project. Nevertheless, Figure 33 shows an early prototype that has been built to initially prove the concept. It includes a rather small display, two buttons, a full size USB connector, and a smart card reader slot for ID-0 (small card as used in mobile phones) sized chip cards.



Figure 33. ZTIC device early prototype

Figure 34 shows the latest prototype, which can be produced in high volumes. It has a larger and brighter display, a mini USB connector, two large buttons, and a wheel at the right side. The additional wheel can be used similar as with combination locks to input a PIN code, if required. Furthermore, the device includes a smart card reader slot (on top, not visible in the figure), which supports ID-1 (credit card size) sized chip cards.



Figure 34. ZTIC device production prototype

4.4. INTEGRATION WITH ePVM

To add support for secure multi-party transaction authorization to the ePVM framework, the ZTIC concept has been integrated with ePVM as an optional ePVM extension. This extension, in principle, implements the second channel usage scenario as presented in Section 4.1.3. Since the second channel approach bears resemblance to the mTAN solution discussed in Section 2.2.5, the ePVM extension is named *ZTAN (ZTIC Transaction Authentication Number) extension*. The ZTAN extension provides the following core functionalities:

1. An ePVM process instance can send transaction requests to be authorized by a particular user to the ZTAN extension.
2. The ZTAN extension manages secure connections with one or more ZTIC devices and forwards pending transaction requests to the ZTIC devices of corresponding users as soon as the devices are connected. The prototype implementation of the ZTAN extension uses EMV CAP (c.f. Section 2.2.6) compliant chip cards for carrying out client authentication when setting up the secure connections. Support for other authentication mechanisms could be added if required.
3. The ZTAN extension reports the results of transaction requests back to the ePVM process instances. Results indicate if the user has authorized or declined a transaction request.

Figure 35 illustrates how the ZTAN extension can be used to implement multi-party transaction authorization. It is assumed that each user is equipped with an EMV CAP

card that securely stores his or her credentials. The card stores a symmetric cryptographic key and it is able to calculate cryptograms for given data using this key. In such a configuration, a ZTIC device does not need to store any secrets to personalize it for a specific person, only inserting the chip card personalizes it temporarily. In other words, people might share ZTIC devices. As indicated in Figure 35, users can activate their ZTIC device by connecting it to any, potentially untrustworthy, client PC, starting the zticproxy, and inserting their chip card. The ZTIC then establishes the security channel (solid arrows in Figure 35) to the ZTAN extension by establishing a TLS connection including server authentication and running the client authentication protocol involving the CAP card. The latter requires the server to send a random challenge for which the card calculates a corresponding cryptogram to prove authenticity. All this happens transparently for the user. Once the security channel is established, pending transactions are retrieved from the ZTAN extension and presented to the user for approval. It is to be noted that users may or may not maintain an information channel (dashed arrows in Figure 35) between a client application running on their PCs and the ePVM host application. Essentially, the ZTAN extensions allows ePVM process instances to securely request authorization for certain transactions from end users without having to care about communication, security, and cryptography. Based on this functionality, it is very easy to define arbitrarily complex multi-party transaction authorization flows via one or more ePVM process definitions. For example, assume an employee submits an order via an order management system that is accessed via a Web browser running on the employee's PC. Furthermore assume, for the order to be processed, two authorizations secured by the ZTIC are required, one from the employee and one from his or her manager. Obviously, the employee would immediately approve the transaction using his or her ZTIC connected to the PC; however, the manager might be travelling. Let us assume at some later point in time the manager arrives at the airport and activates his or her ZTIC by attaching it to a public internet PC. He or she can then securely retrieve, review and approve the pending transaction on the ZTIC without requiring any additional client application.

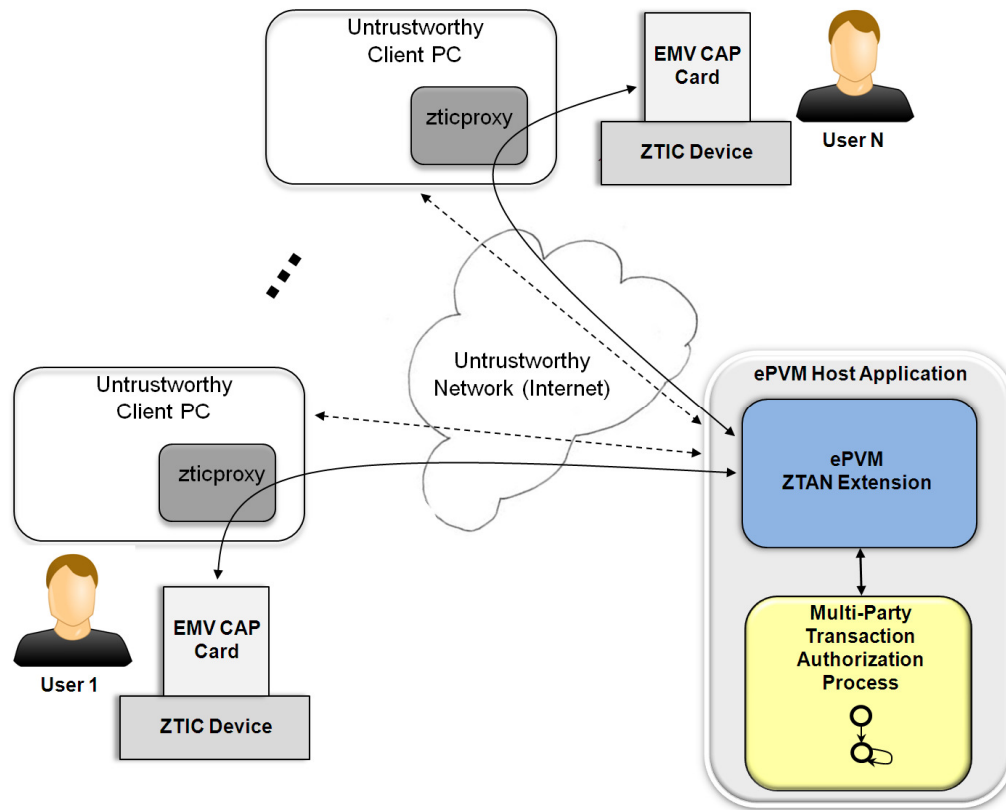


Figure 35. ePVM multi-party transaction authorization scenario

Figure 36 illustrates in more detail how the ZTAN extension is implemented. From an ePVM perspective, it consists of a host process object registered with the engine and thus available for message passing. The ZTAN host process object provides the following functionalities by processing a number of well-defined messages:

1. Add or remove a user. A user is represented by a userID, the ID of the associated EMV CAP chip card, and the corresponding cryptographic key.
2. Submit a transaction authorization request, synchronously or asynchronously, for a given userID. The request consists of the text to be displayed on the users ZTIC device. Since the ZTIC has a two line display, two lines of text can be provided.
3. Cancel a pending transaction authorization request.

Information on users and pending transactions are stored in a database internal to the ZTAN extension. Furthermore, the extension consists of a SSL/TLS Web server handling the connections with the ZTIC devices. The Web server is a standard Apache server supporting SSL/TLS and PHP (PHP: Hypertext Pre-processor). It serves a PHP script that handles authentication and transaction authorization for each ZTIC connection. More precisely, if a ZTIC device connects via TLS, the PHP script sends a random challenge to

the ZTIC and verifies the response to implement client authentication. The relevant cryptographic key is retrieved from the database based on the card ID sent by the ZTIC. Finally, the PHP script regularly checks with the database for pending transactions and forwards them to the ZTIC, if any. Results are written back to the database. In other words, the synchronization between the ZTAN Web server and the host process object is implemented via the database (c.f. Figure 36). In addition to integrating the ZTIC service with ePVM, the ZTAN extension generally demonstrates how any additional functionality can be attached to ePVM via the use of host process objects.

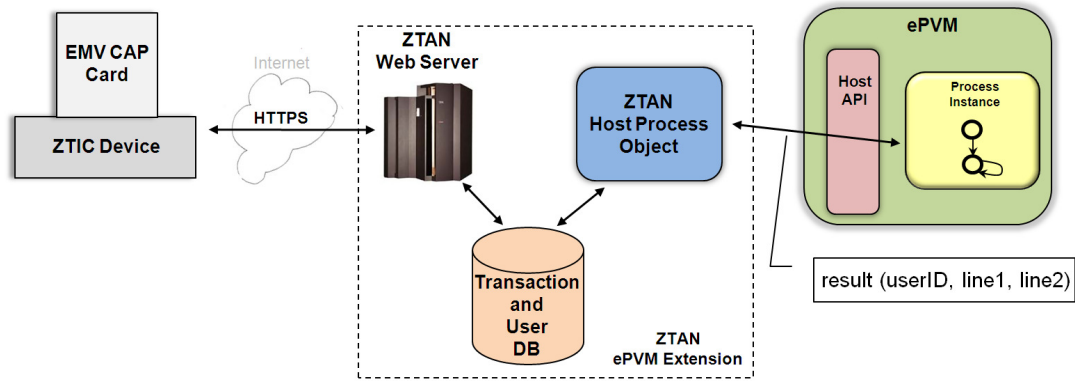


Figure 36. The ZTAN ePVM extension

4.5. SUMMARY AND CONTRIBUTIONS

The ZTIC, as introduced in this chapter, represents a novel approach to solve the problem of secure remote authentication and transaction authorization. It can be used in various application scenarios ranging from internet banking to multi-party transaction authorization as required in business processes. Essentially, it consists of a small USB device including display and buttons that is able to establish an end-to-end TLS channel to a remote server. This way, authentication can be carried out seamlessly and transactions can be securely reviewed and approved or declined on the device itself. While providing the highest level of security, the ZTIC uniquely addresses other aspects such as ease-of-use, mobility, administration, and integration. Finally, the ZTIC approach has been integrated into the ePVM framework to enable the definition of multi-party transaction authorization process flows within ePVM. Table 3 summarizes the main challenges and briefly reiterates how the ZTIC approach solves these. This chapter is based on work published in [50,20,107,21] and several related patents (c.f. Section *Declaration*).

Table 3. Summary: Challenges and ZTIC solutions

Challenge	ZTIC Solution
Authentication, integrity, confidentiality	Standard TLS protocol. Client authentication might be carried out on application level to support different authentication methods
Protection from Phishing attacks	Automatic session establishment with securely stored credentials
Protection from man-in-the-middle attacks	Mutually authenticate end-to-end TLS session between ZTIC and remote server
Protection from malicious software attacks	Display of critical data on trusted device (secure display); Approval via trusted device (secure keypad/buttons)
Non-repudiation	Use of smart card slot and PIN entry on ZTIC (optional)
Ease-of-use and mobility	Small, well-known (key fob) from factor; no software/driver installation; works on all operating systems; easy and fast user interaction
Administration and integration	Few requirements, standard HTTP and TLS protocols only; few server side changes required; secure remote firmware update and configuration; simple standard hardware leading to low cost device

5. CASE STUDY AND EVALUATION

This penultimate chapter of this thesis presents a series of experimental scenarios established for basic evaluation of the ePVM framework and the ZTIC solution. Afterwards, a case study used to evaluate the results of this thesis in a comprehensive process-driven application is presented. This study has been carried out in the context of the GridCOMP research project funded by the European Union. It is used to validate the architecture as well as the prototype of the ePVM framework in a real world scenario – a distributed biometric identification system. Results, experiences, and lessons learned are summarized with the focus on the integration of ePVM and autonomic Grid components used for building the distributed application. Finally, ZTIC support has been added to the application to demonstrate and evaluate its properties in the context of a complex application.

5.1. ePVM FRAMEWORK

5.1.1. Basic Framework Validation

Prior to using the ePVM framework in the context of a comprehensive use case application as described in Section 5.3, its basic properties have been evaluated by means of the stock quote example application as described in Section 3.8.5. Here, the focus was on evaluating the general suitability of the architecture and its basic performance properties. The former has been analyzed with respect to the challenges and requirements described in Section 1.8.1.

Firstly, it can be observed that the ePVM architecture and the corresponding prototype implementation is kept strictly generic and thus potentially supports interoperability. The generic nature has been achieved by focusing on basic functionality such as state and control flow management only and excluding support for any domain specific protocols such as Web services from the core engine. As a result, the use of ePVM did not pose any notable requirements such as a server environment on the stock quote application. Nevertheless, the stock quote application demonstrates that support for Web service communication can be easily attach to the ePVM engine via the host process object interface. Table 4 shows the source code break down for the software components that constitute the stock quote example. The fact that the host process object implemented to communicate with the stock quote Web service consists of not more than 63 lines of

source code emphasises the suitability of the architecture and the relevant host API. The same applies to the monitor and alert component, which are equally lean. Additionally, the fact that it is not necessary to learn a new process definition language significantly reduced the effort for building the first ePVM based application. Ultimately, demonstrating interoperability would involve mapping two or more domain specific process languages to the ePVM model. Although this is subject to future work, it can be assumed that the JavaScript language in combination with the CEFSM approach provides the expressive power for defining such mappings.

Table 4. Stock quote example source code break down

Component	Lines of Code
Host application	128
Process definition	49
Web service host process object	63
Monitor object incl. GUI	122
Alert host process incl. GUI	45

Secondly, it can be observed that the prototype implementation is lightweight and easily embeddable. What makes it lightweight is the combination of size and requirements. The prototype ePVM library has a size of about 60 kilobyte (KB) and the only requirements are the availability of the Rhino JavaScript library, which is about 700 KB, and a standard Java runtime environment. Consequently, the stock quote application could be conveniently developed as a standard Java application and it could be packaged together with ePVM and Rhino into a platform independent application of 800 KB.

Finally, it can be observed that the simplicity of JavaScript combined with the CEFSM-based programming model provides a very easy and powerful way of programming concurrent control flows and synchronization. As a result, the process definition for the stock quote application (c.f. Listing 14) is, with a size of 49 lines of code, very compact and, due to the popularity of JavaScript, quite easy to understand for a large number of people. The fact that ePVM is directly programmable in such an easy way makes it unique compared to other process engines, which either do not support direct programming (without any high-level tools and/or definition languages) at all or provide a quite complex object model in a full blown object oriented programming language. The example also illustrates the benefits of the process-driven application approach. Imagine

implementing the control flow of the application directly in a standard programming language using threads, semaphores, object monitors, or similar primitives to control concurrency and synchronisation. The code would not only be less flexible, because it would be compiled rather than interpreted, but also more complicated and error prone.

The evaluation of the stock quote application with respect to core requirements such as versatility and programmability shows that the ePVM architecture is suitable for building process-driven applications in an efficient way. However, besides having a suitable architectural design, programming model, and corresponding APIs, it is equally important that the system performs reasonably well in practice. Thus, some basic execution performance properties of the prototype implementation have been measured and assessed. Again, the stock quote application was used and instrumented to measure certain properties. In a first experiment, the stock quote application was configured to monitor 200 stocks concurrently whereas the quote for each stock was retrieved from the Web service every 30 seconds (c.f. process flow as in Figure 29). This means that 200 instances of the stock quote ePVM process were executed by the engine concurrently. As the focus was on the execution performance of the ePVM engine, the Web service host process as well as the monitor object was modified for the experiment. The former was modified to not actually connect to the remote Web service but just return a fixed stock price and the latter was modified to not visualize the stock prices in the GUI as both operations are time and performance intensive. This way, it was possible to measure the approximate CPU and memory usage of the ePVM engine while executing the 200 process instances by monitoring the application via the Java management console. Figure 37 shows the CPU usage while running the application for 10 minutes. Initially, the 200 process instances were created and triggered. From then on, each instance was idle for 30 seconds, then triggered the Web service, compared the stock price, triggered the monitor, and finally became idle for another 30 seconds. The CPU usage diagram reflects this behaviour. Every 30 seconds the 200 processes were woken up by their timers and the ePVM runtime required about 10% CPU to execute them. As a result, the diagram shows a CPU usage peak every 30 seconds. Figure 38 shows the corresponding Java heap memory usage during the 10 minutes. From an initial value of around 2 megabyte (MB), the memory usage went up to around 5.5 MB as soon as the 200 process instances were created. During the experiment, the memory usage oscillated around 5.5 MB depending on the activity of the Java garbage collector. The quite low CPU usage shows that the

ePVM runtime can easily handle 200 concurrent process instances. Furthermore, it can be seen that the internal data structures comprise not more than 3 MB for the 200 processes. Although the ePVM prototype has not been optimized for performance and resource consumption, it seems to be quite efficient. The experiment also shows that the single threaded nature of the core engine does not represent a bottleneck if the processes are defined in a way such that the real work is implemented in host process objects executed by OS threads. The experiments were run on a standard PC equipped with an Intel Core 2 Duo 2.4GHz CPU running Java 1.6.

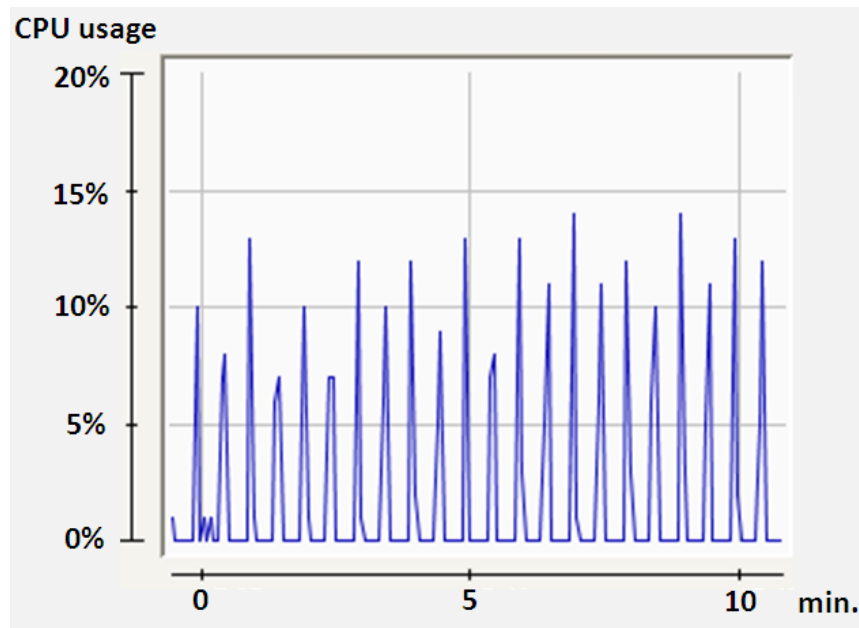


Figure 37. CPU usage executing 200 stock quote instances

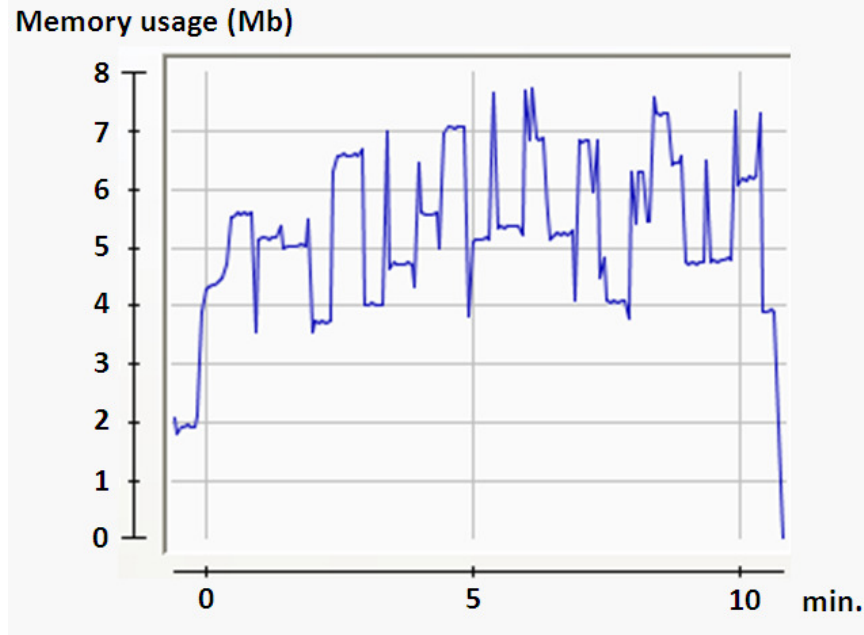


Figure 38. Memory usage executing 200 stock quote instances

In addition to the overall execution performance, some more fine-grained measurements have been carried out to determine the execution performance of certain basic operations within ePVM. For this, the stock quote application was further modified to collect runtime measurements using the *nanoTime()* method available in Java. Again, measurements were carried out using the hardware mentioned earlier. Each measurement was repeated ten times and results were averaged. Table 5 shows the execution times for six important basic operations. Operations 1 and 2 are triggered by the host application via the host API. The other operations represent message roundtrip times between the involved entities, namely, the host application, process instances, host process objects, and monitor objects. Roundtrip time means that the sending entity, for instance, the host application in operation number 3, sends a synchronous message and the receiving entity immediately sends a reply message without any additional message processing. Table 5 shows that the ePVM prototype performs quite well. Operation 4 is the fastest as it does not involve external entities and thus does not require communication and synchronization with OS threads. Nevertheless, it involves two context switches, firstly for activating the receiving process instance, and secondly for switching back to the sending process instance. A performance of about 1.2 msec shows that the message-driven threading system of ePVM is efficient considering the fact that it is implemented on JavaScript level. Yet, further investigations have shown that the implementation of continuations in the Rhino interpreter leaves room for optimizations. Overall, it can be

noted that the execution performance of the ePVM prototype is sufficient for validating and demonstrating the ePVM approach. Performance optimization and subsequent detailed evaluation including the comparison of scenarios with other execution engines are subject to future work. In general, it can be assumed that an interpreter-based approach always stays behind compiled code with respect to execution performance.

Table 5. Execution performance of basic operations

	Operation	Execution Time (μ sec)
1	Deploy process definition (via host API)	35686
2	Create process instance (via host API)	16357
3	Message roundtrip (host application \leftrightarrow process instance)	4119
4	Message roundtrip (process instance \leftrightarrow process instance)	1244
5	Message roundtrip (process instance \leftrightarrow host process object)	1921
6	Monitor roundtrip	1669

5.1.2. Persistence

This sub-section presents an experiment carried out to evaluate ePVM's persistence model introduced in Section 3.7. In this experiment, the goal was to define a sample process, which demonstrates the flexibility of the model and the effectiveness with respect to resource consumption. At the same time, it should simulate common properties of real-world business processes. The properties that have been focused on are:

1. Many instances of a process exist concurrently.
2. The process stores some application-level data as part of its state.
3. The process is long running (it becomes, at some point during its lifetime, idle for a rather long period of time).
4. When becoming idle, the process decides, based on application-level data, whether to request persistence via a particular persistence provider or not.
5. Most of the process instances are idle at any given point in time.

A real-world example of a business process including these properties would be an order management process. Typically, a large number of orders exist concurrently and it takes a rather long time until an order is processed completely. This is due to the fact that at

some points the process has to wait for long-running activities such as payment or shipment to be completed. As a result, the process is idle most of its overall lifetime.

```
// initial message - unique process ID (value 0-n)
function persistence__PVMPProcess(environment, message) {
  if (environment.state == null) { // initial message
    PVM_call(PVM_Timer, {"mode":"duration", "msec":120000});
    environment.state = new String();
    for (var i=0; i < 1024; i++) // allocate 20 kB string
      environment.state += "FFFFFFFFFFFF";
    PVM_call(PVM_Timer, {"mode":"duration", "msec":120000});

    PVM_send(PVM_Timer, {"mode":"duration", "msec":120000});
    if (message > 100)
      PVM_persist(PVM_hostProcess("file persistence"));
    return true;
  }
  else { // asynchronous timer expired
    PVM_call(PVM_Timer, {"mode":"duration", "msec":120000});
    return false; // terminate process
  }
}
```

Listing 15. ePVM process simulating real-world business process

For the experiment, the ePVM process shown in Listing 15 was defined, which simulates the properties mentioned earlier. Initially, it receives a message carrying a unique ID. Upon receiving this message, it simulates the allocation of 20 KB of application-level data. Before and after the allocation a synchronous timer is set up (via *PVM_call()*) to block execution for two minutes. This is just to make the data allocation clearly visible in a memory usage trace later on. Then, the process becomes idle for two minutes by setting up an asynchronous timer (via *PVM_send()*) and returning from the process function. Before returning, however, it requests to be stored persistently via the persistence provider named *file persistence* if the ID value is larger than 100. Once the asynchronous timer expires, the process is triggered again receiving the timer message. Then it sets up another synchronous timer and finally terminates. In other words, the basic steps of the process are (all timers run for two minutes): Timer 1, Allocate data, Timer 2, Process becomes idle and is stored persistently, Timer 3, Process is restored from persistent memory, Timer 4, Process terminates.

For running the experiment, a test program was used that deploys the process definition to the ePVM engine, creates 1000 process instances, and then sends an initial message to each instance carrying a unique ID value (0-1000). After starting the experiment, the memory usage of the process engine during the lifetime of the process instances was recorded. As the engine is implemented in Java, it relies on the memory management of the underlying Java Virtual Machine (JVM). Therefore, the heap memory usage of the

JVM was recorded while regularly triggering garbage collection to ensure the heap size reflects the actual memory usage. Figure 39 shows the resulting graph illustrating the Java heap memory usage in MB over ten minutes. The graph shows that starting from the initial memory usage level L0, after about 30 seconds, the test program starts creating the process instances. Once all 1000 instances have been created, the memory usage level L1 at about 9 MB has been reached. Then, at about 2.5 minutes, the first timer expires and all instances start allocating the 20 kB of state data. Afterwards, the usage level L2 at about 40 MB has been reached. This represents the memory requirement for the case where the process does not take advantage of persistence. Once the second timer expires, 900 out of the 1000 process instances request to be stored persistently and become idle. This simulates the situation where with long-running processes most instances (here 90%) are idle at any point in time and thus they can be removed from transient memory. As a result, memory usage decreases significantly until it reaches level L3. The reason why L3 is lower than L1 although 100 process instances are still held in transient memory is the fact that not only the process states, including the 20 kB of data, but also the related threads are released. Then, after the asynchronous timer has expired, the process instances are automatically restored from persistent memory and consequently the memory usage level L2 is reached again. Eventually, after the final timer expires, the process instances terminate such that the memory usage level drops to its initial level L0.

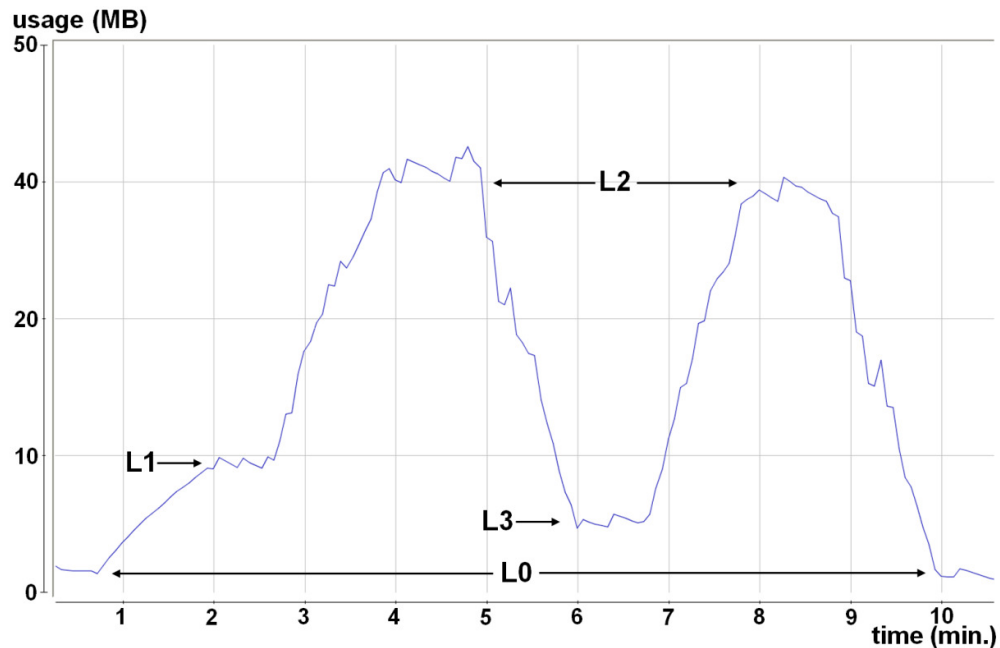


Figure 39. Java heap memory usage during experiment

A number of conclusions can be drawn from this experiment. Firstly, it demonstrates that the flexible persistence model makes it very easy for individual process instances to decide at runtime whether they should be stored persistently or not based on application-level data. In the experiment, we simply used the ID value to decide about this. In the corresponding real-world example, the order management process could, for instance, decide based on the payment method chosen by the customer. Secondly, the results clearly show that storing idle processes persistently can significantly reduce resource consumption. In the experiment, we could observe memory consumption being reduced by approximately 87.5% (L2 vs. L3) when assuming that only 10% of the process instances are required to be in transient memory concurrently. Finally, the experiment also indicates that storing and restoring the process instances requires quite some computing time. Further measurements unveiled that the average time required to store/restore a process instance is 77/36 milliseconds. Of course, these numbers depend on the implementation of the model in the process engine, the persistence provider used, the hardware used, and so forth. Anyhow, when optimizing resource consumption through the usage of persistence, the trade-off between the process idle time, the performance of the persistence sub-system, and the load of the machine running the process engine clearly must be considered. To be able to do this, ideally dynamically at runtime, a flexible persistence model like the one implemented in ePVM is essential.

5.2. THE ZTIC

The initial evaluation of the ZTIC approach was carried out by means of a test application implementing an Internet bank. As outlined in Figure 40, the application consists of an SSL/TLS enabled Web server, which hosts two Web applications, one representing the online portal of the Internet bank, and the second managing connections to ZTIC devices operating in second channel mode. Both applications interact via a shared database storing information about transactions and users. Through the portal, a user can log in to the bank by providing user ID and password. Afterwards, a login confirmation request is sent to the user's ZTIC device such that he or she can confirm the login operation via the ok button on the ZTIC. Once logged in, the user can navigate to a Web page displaying a payment form. As soon as the user submits this form, the payment details are sent to the user's ZTIC for review and confirmation. Only if the user confirms via the ZTIC, the payment transaction is considered as submitted and ready for processing.

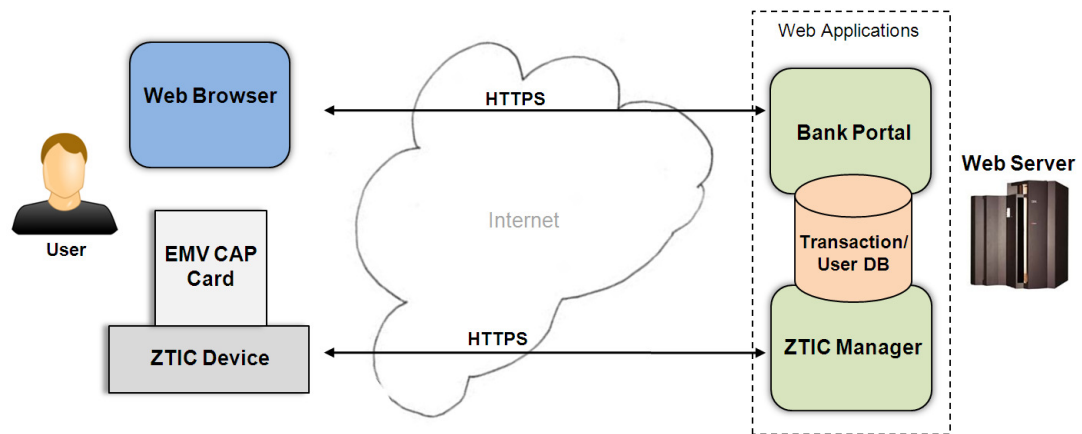


Figure 40. ZTIC internet bank application

The simulated Internet bank application using the ZTIC for authentication and transaction authorization was used to evaluate the ZTIC with respect to the challenges and requirements defined in Section 1.8.2. Experiences and conclusions drawn from this test and demo application can be summarized as follows:

1. With respect to integration, building the Internet bank application showed that the ZTIC approach integrates exceptionally well with today's standard technologies. For the Web server, the Uniform Server [108], which integrates Apache, MySQL, and PHP into one packaged solution, was used. As a result, the only efforts necessary to develop the application where: configure appropriate SSL/TLS certificates on the server and the ZTIC device, develop the two Web applications by means of a few PHP and HTML pages that set up and use the built-in MySQL database. In other words, standard Web application development tools and skills were sufficient to integrate the ZTIC into the application.
2. With respect to ease-of-use, the Internet bank application was tested by more than 20 people including researches as well as their families. The general feedback was that initially it was not clear to all users when they should move attention from the Web browser to the ZTIC device and vice versa. In addition, locating and starting the zticproxy turned out to be a challenge for a few users. However, all issues appeared only during first time use. Once, the users got used to the ZTIC interaction patterns, they where very happy with the ease-of-use. Some of

the users even questioned security because the convenient usage pattern made them feel that the solution might not be secure.

3. Security against software attacks is guaranteed by design. However, the issue of social attacks remains an open issue. The experiment has shown that it is critical and non-trivial to educate users towards fully relying on the information displayed on the ZTIC device.
4. In general, mobility is also guaranteed by design. Users were able to access the Internet bank from various different client PCs without installing any software. They particularly liked the small form factor of the device. However, it turned out that some users have support for USB MSD disabled by their system administrators and thus the ZTIC cannot be used at all.

Overall, the first real-world ZTIC application showed that the approach satisfies the requirements defined beforehand.

5.3. THE BIOMETRIC IDENTIFICATION USE CASE

5.3.1. Introduction

At the time of writing, process-driven applications are becoming increasingly popular in the industry. The main advantages of this approach are the fact that the application logic can be modified without re-compiling the application, the business logic is more evident, and monitoring features of the process engine can be exploited. Besides the trend towards process-driven applications, enterprises seek ways to benefit from resources available from computing Grids/Clouds, in particular, in all those cases where parallel computing is required to guarantee fair performances. The development of grid applications is not an easy task, however. Grid architectures present peculiar features such as dynamicity, heterogeneity, and non-exclusive access to resources that require substantial effort to be suitably handled. Within the plethora of programming environments targeting Grids, the Grid Component Model (GCM) developed within CoreGRID [109] and whose reference implementation has been provided by GridCOMP [31], supports Grid programmers in designing parallel/distributed grid applications. In particular, GCM provides pre-defined composite components modelling standard parallel/distributed computation patterns that users can instantiate by just providing the components implementing the sequential computations involved in the parallel pattern. In addition, these pre-defined composite components implement proper autonomic managers that completely take care of non-

functional aspects related to application execution according to what is specified in user-supplied contracts.

In Section 5.3, a process-driven application is presented, which makes use of GCM autonomic components to solve the problem of large-scale biometric identification. The application has been developed as part of the activities of the GridCOMP project. With respect to the GridCOMP project, the application represents a use case for demonstrating and evaluating the GCM framework. Additionally, with respect to this thesis project, it represents a use case for demonstrating and evaluating the results of this thesis, as the application has been designed as a process-driven application based on the ePVM framework. In particular, it provided the opportunity to evaluate how the ePVM based process-driven approach integrates with the Grid component framework.

5.3.2. The Grid Component Model Framework

5.3.2.1. Overview

The *Grid Component Model* (GCM) is a component model explicitly designed to support component-based autonomic applications in distributed contexts. The main features of this component model can be summarised as follows:

- **Hierarchical:** GCM components can be composed in a hierarchical way in *composite* components. Composite components are first class components and they are not distinguishable from non-composite components at the user level. Hierarchical composition greatly improves the expressive power of the component model and is inherited by GCM from the Fractal component model [110].
- **Structured:** In addition to standard intra-component interaction mechanisms (use/provide ports [111]) GCM allows components to interact through *collective* ports modelling common structured parallel computation communication patterns. These patterns include broadcast, multicast, scatter and gather communications operating on collections of components. Also, GCM provides *data* and *stream* ports, modelling access to shared data encapsulated into components and dataflow streams. All these additional port types, not present in other well known component models, increase the possibilities offered to the component system user for developing efficient parallel component applications.
- **Autonomic:** GCM specifically supports implementing autonomic components in two distinct ways: by supporting the implementation of user defined component

controllers and by providing behavioural skeletons. Component controllers can be programmed in the component membrane (the membrane concept, as the place where component control activities take place, is inherited from Fractal) and controllers can be components themselves. This provides a substantial support to the development of reusable autonomic controllers. Behavioural skeletons, thoroughly discussed in Section 5.3.2.2, are composite GCM components modelling notable parallel/distributed computation patterns and supporting autonomic managers, for instance, components taking care of non-functional concerns affecting parallel computation.

Due to the presence of controllers and autonomic managers, GCM components implement two distinct kinds of interfaces: functional and non-functional ones. The functional interfaces host those ports concerned with the implementation of the functional features of the component. The non-functional interfaces host the ports related to controllers and autonomic managers. These ports are the ones actually supporting the component management activity in the implementation of the non-functional features, for instance, those features contributing to the efficiency of the component in obtaining the expected (functional) results but not directly involved in result computation.

5.3.2.2. *Behavioural Skeletons*

Behavioural skeletons (BS) represent a specialisation of the algorithmic skeleton concept for component management [112]. Algorithmic skeletons have been traditionally used as a vehicle to provide efficient implementation templates of parallel paradigms. BS, as algorithmic skeletons, represent patterns of parallel computations, which are expressed in GCM as graphs of components, but in addition, they exploit the inherent skeleton semantics to design sound self-management schemes of parallel components. BS are composed of an algorithmic skeleton together with an autonomic manager as outlined in Figure 41. They provide the programmer with a component that can be turned into a running application by providing the code parameters needed to instantiate the algorithmic skeleton plus some kind of Service Level Agreement (SLA) sometimes also referred to as the Quality of Service (QoS) contract, for instance, the expected parallelism degree or the expected throughput of the application. The code parameters are used to build the actual code run on the target parallel/distributed architecture, while the SLA is used by the autonomic manager that will take care of ensuring it while the application is being computed.

The choice of the skeleton to be used as well as the code parameters provided to instantiate the BS are functional concerns: they only depend on what has to be computed and on the qualitative parallelism exploitation pattern the programmer wants to exploit. The autonomic management itself is a non-functional concern. The self-management and self-tuning activities taking place in the manager to ensure user supplied SLA both depend on the application structure and on the target architecture at hand. The implementation of both the algorithmic skeleton and the autonomic manager is in the charge of the system programmer, for instance, the one providing the behavioural skeleton framework to the application user. In general, the application programmers are in charge of picking up a BS, or a composition of BS, among those available and of providing the corresponding parameters and SLA. The system, and in particular the autonomic managers of the behavioural skeletons used, are in charge of performing all those activities needed to ensure the user supplied SLA.

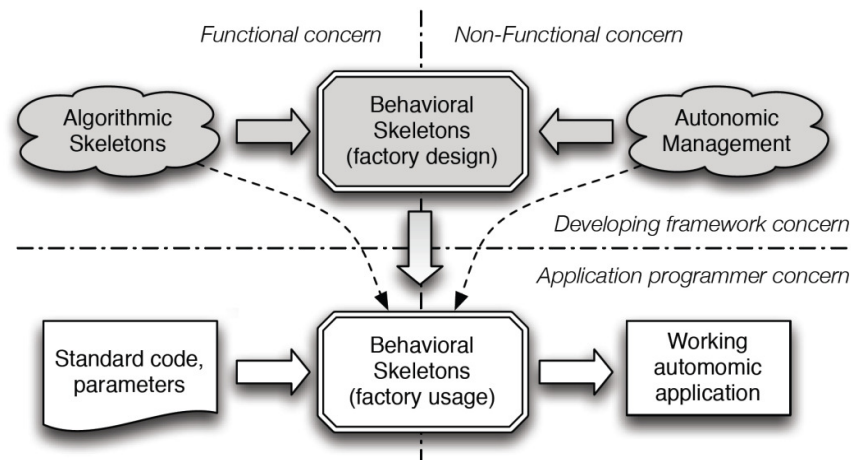


Figure 41. Behavioural skeleton rationale

Autonomic management of non-functional concerns is based on the concurrent execution (with respect to the application logic) of a basic control loop consisting of a monitor, analyze, plan, and execute phase. In the monitor phase, the application behaviour is observed, then in the analyse and plan phases the observed behaviour is examined to discover possible malfunctioning and corrective actions are planned. The corrective actions are usually taken from a library of known actions and the chosen action is determined by the result of the analysis phase. Finally, the actions planned are applied to the application during the execute phase [113,114].

At the time of writing, two kinds of behavioural skeletons are implemented in GCM: a *task farm* BS and a *data parallel* BS as shown in Figure 42. The former models

embarrassingly parallel computations processing independent items x_i of an input stream to obtain items $f(x_i)$ of the corresponding output stream. The latter models data parallel computations by computing for each item of the input stream x_i an item $f(x_i;D)$ of the corresponding output stream, where D represents a read only data structure and the result of $f(x_i;D)$ can be computed as a map of some function $f_0(x_i)$ on all the items of D followed by a reduce of the different $f_0(x_i;D_j)$ with an associative and commutative operator g . Both BS implement an autonomic manager (AM) taking care of the performance of the parallel computation at hand. In particular, the AM may ensure contracts stating the expected service time or throughput of the task farm BS, or the expected partition size of data structure D in case of the data parallel BS. In the GridCOMP prototype, the contracts must be supplied to the BS AMs through BS non-functional ports as a string hosting a set of JBoss rules defined in terms of the operations provided by the autonomic behaviour controller (ABC) controller. In fact, the AM control loop is implemented by running an instance of the JBoss business rule engine at regular intervals of time. At each time interval, all the pre-condition-action rules supplied to the AM are evaluated and those that turn out to be fireable are executed ordered by priority. The pre-conditions are evaluated using values provided by the monitoring system implemented in the ABC controller. The AMs manage the contracts by varying the parallelism degree of the BS, meaning the number of worker instances actually used to implement the BS. The variation of the number of worker instances happens by adding/removing a fixed amount of workers. This fixed amount is a BS user configurable constant.

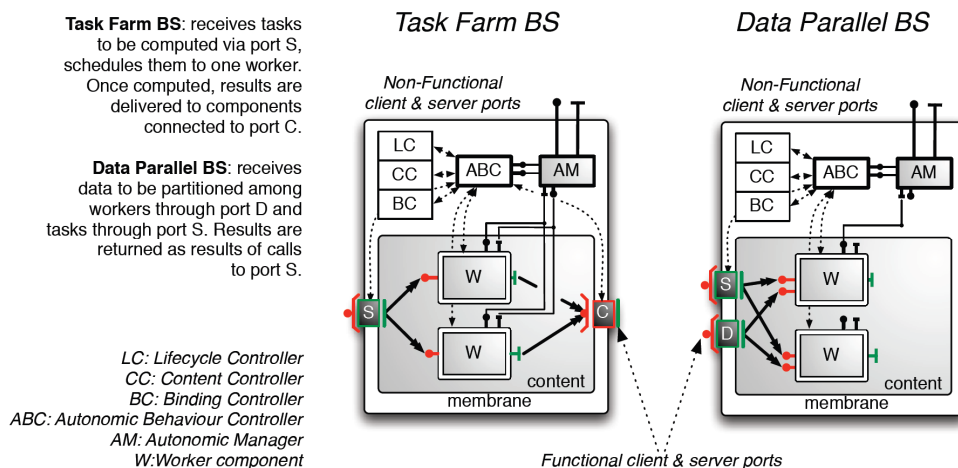


Figure 42. Behavioural skeletons implemented in GCM

5.3.3. Process-Driven Distributed Biometric Identification

5.3.3.1. Introduction

In recent years, biometric methods for verification and identification of people have become very popular. Applications span from governmental projects like border control or criminal identification to civil purposes such as e-commerce, network access, or transport. Frequently, biometric verification is used to authenticate people meaning that a 1:1 match operation of a claimed identity to the one stored in a reference system is carried out. In an identification system, however, the complexity is much higher. Here, a person's identity is to be determined solely on biometric information, which requires matching the live scan of his or her biometrics against all enrolled (known) identities. Such a 1:N match operation can be quite time-consuming making it unsuitable for real-time applications. In order to tackle this challenge, a distributed biometric identification system (BIS), which can work on a large user population of up to millions of individuals, has been developed. It is based on fingerprint biometrics and allows real-time identification within a few seconds period. To achieve this, it takes advantage of the Grid via autonomic GCM components. Furthermore, to ensure adaptability of the application, it is designed as a process-driven application.

5.3.3.2. Application Architecture

The BIS can be considered a process-driven application, as it is centrally driven by the ePVM process engine. Figure 43 outlines its high-level architectural design. A number of ePVM process definitions describing the main control flow for operations such as starting up the system or identifying a person are loaded into the process engine. These processes cooperate with external entities such as the GUI, the database (DB) of known identities, and the distributed GCM component system via a number of host process objects to implement the overall functionality of the BIS. The following sub-sections describe how the BIS works and discuss the challenges faced and experiences made while developing the application. Here, the focus is on the usage of ePVM and its integration with the GCM framework.

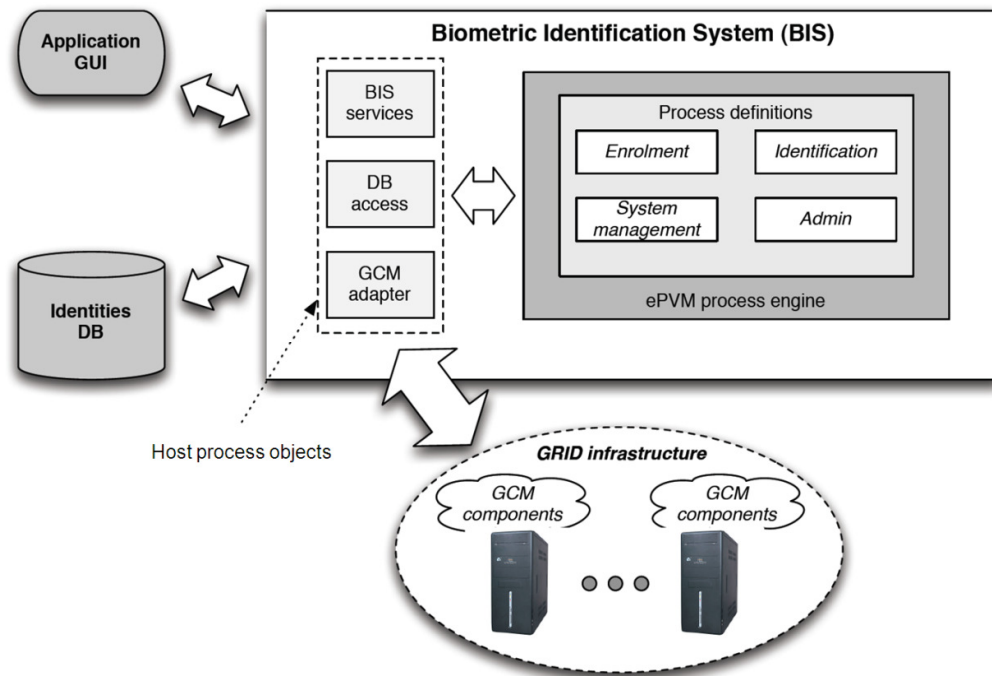


Figure 43. BIS high-level architecture

5.3.3.3. Process Engine/Grid Interfacing

The actual distributed fingerprint matching functionality is implemented via a set of GCM components deployed within a Grid/Cloud infrastructure as indicated in Figure 43. Processes running within the process engine must be able to create, deploy, configure and interact with these components. For this purpose, a dedicated host process object named *GCM adapter* (c.f. Figure 43) has been developed, which receives messages from ePVM process instances, turns these messages into method invocations on GCM framework methods or GCM components, and generates appropriate reply messages returned to ePVM. The GCM adapter represents the main interface between ePVM and GCM. As ePVM process definitions are implemented in JavaScript and the GCM framework is available as a Java library, the GCM adapter essentially converts between JavaScript messages and Java method invocations. An alternative option would have been to export the GCM components as Web services, as supported by the GCM implementation, and invoke them from within the GCM adapter. However, this would have increased the number of required type conversions going from Java Script over XML/SOAP to Java and vice versa. In addition, the GCM framework only supports exporting GCM components as Web services. Other framework services, for example, functionality for deployment and component creation, cannot be turned into Web services automatically.

Finally, the ePVM process engine does not necessarily require working on Web services level as, for instance, process engines based on BPEL. Consequently, it was decided not to use Web services as interfaces between the process engine and GCM. The functionality provided by the GCM adapter includes:

- Activate a given GCM deployment descriptor to start the nodes available in the Grid.
- Modify architecture description language (ADL) files describing the GCM components used.
- Create GCM components within the Grid.
- Invoke methods on GCM components, for example, to configure the QoS contract, distribute the DB of known identities, or submit the biometrics of a person for identification.

The GCM adapter as well as the host process objects (c.f. Figure 43) are triggered by ePVM process instances to implement the overall application logic. As an example, the activity flow chart shown in Figure 44 illustrates the control logic defined within the ePVM process executed during BIS application initialization. For each of the activities a message is being sent to a host process object that implements the corresponding functionality. Some of the activities execute in parallel, for instance, activity 1.1 to 1.3, some are sequential. The corresponding ePVM process definition can be found in Appendix B.

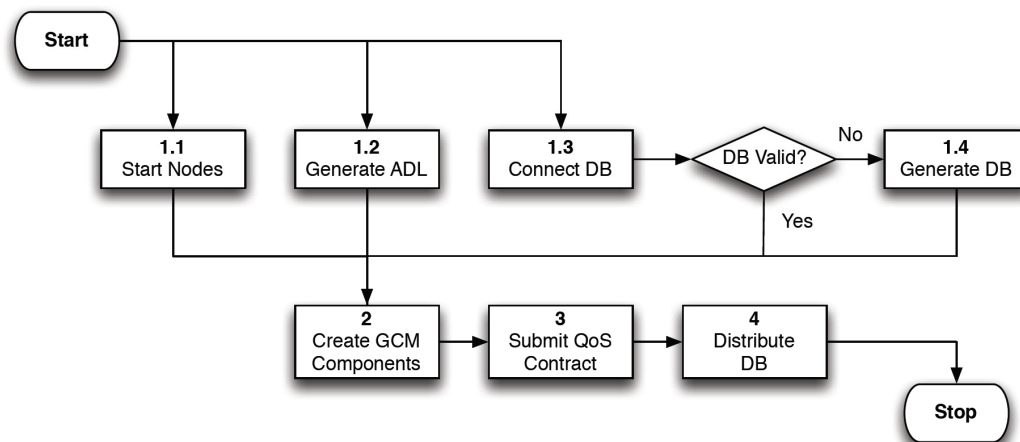


Figure 44. BIS initialization process flow

5.3.3.4. Using Autonomic Grid Components

The problem of biometric identification can be considered a search problem where the compare function is a biometric matching algorithm, here fingerprint matching. To distribute the problem within a Grid infrastructure, the database (DB) of known identities needs to be distributed such that each computing node in the Grid receives a partition of the overall DB and can match a given identity against this partition. Furthermore, the resulting system should reconfigure itself autonomously such that the number of nodes used complies with a given QoS contract. To implement such a system the data-parallel BS has been used. Figure 45 illustrates the GCM component system employed and indicates how the process-driven BIS makes use of it. For the BIS, the skeleton has been parameterized with an appropriate worker component and QoS contract. The worker component, here named *IDMatcher*, implements the actual fingerprint matching functionality and the skeleton allocates one instance of this worker component per Grid node (CPU core). The QoS contract chosen monitors the partition size of the workers and adds or removes one worker if the partition size exceeds or is less than a certain threshold.

Before identification requests can be processed, the identity DB must be initially distributed across the worker components. The identity DB holds information such as name, address, and fingerprints of all enrolled (known) people. For distributing the DB, the skeleton offers a multicast port, also called the scatter or initialization port, which takes a list of tasks as input parameter. Each task references a part of the identity DB by index and length. The skeleton distributes these tasks equally among the workers and the workers load the parts from the shared DB. As a result, each worker can be considered to have one partition of the DB loaded into transient memory as indicated in Figure 45.

Once the skeleton has been initialized, identification requests can be submitted via the second multicast port provided by the skeleton, the so-called broadcast port. Fingerprints of a person to be identified are broadcasted via this port to all worker components and each worker matches them against its partition of the DB. Results are returned synchronously via method return values. If the AM triggers reconfiguration via the ABC, for example, to increase the number of worker components, the ABC retrieves all tasks from all workers, modifies the number of workers, and finally redistributes the tasks. This way the DB is redistributed during each reconfiguration operation.

The dashed arrows in Figure 45 indicate method invocations of GCM components triggered from ePVM processes via the GCM adapter.

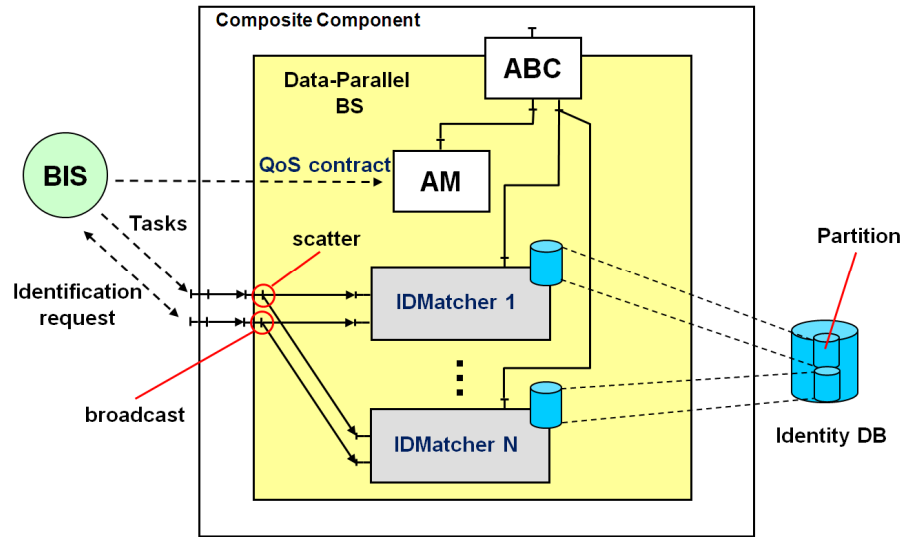


Figure 45. BIS using the data-parallel BS

5.3.3.5. Application Monitoring

Monitoring is one of the core features of every process engine and it is an important argument for using one when building an application. In the BIS application, a process monitor (c.f. Section 3.6) is used to track the progress of ePVM process instances, for example, while the system initialization process is executed (c.f. Figure 44). The process monitor is triggered by the process engine whenever activities start or finish and it updates the GUI to reflect the state of the system.

Furthermore, it was desired to monitor the GCM component system with the goal to visualize AM actions and the number of workers used in the data-parallel skeleton. For monitoring the skeleton, functionality provided by the GCM framework can be used. However, monitoring in GCM is based on a pull model where information about components and their states can be retrieved on request. On the contrary, the ePVM approach can be considered a push model where a monitor is registered and receives events. To integrate GCM monitoring with the event-driven paradigm applied in ePVM some adaptation is necessary. A first solution is to create a dedicated ePVM process, which regularly retrieves information about the component system via the GCM API and creates events for the monitor object. A second solution is to instrument the component implementation to actively send events to an ePVM process. The first approach is more generic with respect to distribution as the GCM framework handles remote method

invocations required to query for component states automatically. The second approach is more efficient as communication only takes place if an event to be monitored occurs. However, a component might not be able to easily communicate with the process engine if it is running on a remote machine, since the process engine itself is not a GCM component. In the BIS the first approach has been to implement monitoring the number of workers, as the workers are typically distributed. For monitoring AM actions, the second approach has been used exploiting the fact that in concrete deployments the AM is always co-located with the process engine such that no remote communication is necessary.

In general, the requirement to monitor actions within the BS to some extent is contradictory to the idea of autonomic components. On one hand, the goal of using the skeleton is to take advantage of its built-in functionality without taking care of the implementation details. On the other hand, it is desired to be able to monitor certain internal details such as reconfiguration operations and the number of workers. From the perspective of the process-driven applications paradigm, all actions that shall be monitored should be centrally controlled by the process-engine. However, in real-world applications a trade-of between central process control and autonomy must be made.

5.3.3.6. *Autonomic Futures vs. Message Passing*

When integrating process-engines and distributed computing frameworks, it is very important to be aware of their communication and synchronization paradigms. The GCM framework is based on Java RMI and implements the concept of automatic futures [115]. This means that method invocations always return immediately, whereas results which are not yet available are represented by so-called future objects. Program execution is then blocked automatically if a future object is being accessed as long as the value represented is not yet available. The goal is to ease parallel programming by hiding synchronization details within a meta-object protocol implemented in GCM. The ePVM process engine, however, uses message passing for communication and synchronization between concurrent control flows. If these two paradigms are interweaved, as it is the case in the BIS application, process flows can easily become distorted. For example, if a process definition assigns two activities to be carried out sequentially (c.f. activity 2 and 3 in Figure 44), it must be ensured that no more future objects resulting from the first activity exist before the second is triggered. This issue becomes obvious when identification is triggered within the BIS. In this case, an ePVM process sends a message

to the GCM adapter including fingerprints of a person to be identified. The GCM adapter forwards this information to the component system by invoking the broadcast interface of the BS (Figure 45). This interface is a collective interface, which turns one method invocation into N method invocations on all the bound IDMatcher components to broadcast the identification request. The return value is a list of result objects, one from each IDMatcher component. When the interface is invoked, it immediately returns a list of future objects, which at the beginning are all unavailable and then by-and-by become available as the IDMatcher components return their results. It is important that the GCM adapter waits for the futures to become available and generates messages to be returned to the ePVM process instance accordingly. It must not report the identification as completed before all futures are available. Effectively, the GCM adapter retracts automatic synchronization in order to make the actual progress visible to the process engine, which must be informed whenever an IDMatcher component has searched its part of the DB. Obviously, converting from one paradigm to the other must be handled with care as the semantics of process definitions can be broken due to delayed synchronization within GCM.

5.3.3.7. Securing Transactions via the ZTIC

The most sensitive operations in the BIS are adding/removing a person to/from the DB of known identities. If an attacker manages to do this, the security of the system is broken completely. For instance, assume a terrorist could remove himself from the terrorist DB such that he can freely travel without being detected at boarder control. Therefore, these highly sensitive operations have been secured via strong multi-party transaction authorization using the ZTIC. This means, whenever a person is to be added or removed from the BIS DB, two people, the BIS operator as well as the chief security officer, need to approve this transaction via their ZTIC devices. Only then, the BIS actually processes the add/remove request. The prototype ZTIC devices along with the ZTAN extension presented in Section 4.4 made the implementation of such secure BIS DB transactions easy. The ePVM process that is controlling the add/remove operation simply requests approval from the two persons via the ZTAN extension. Only if both approve, the process triggers the DB host process object to add/remove the identity to/from the DB.

5.3.4. Results, Experiences, and Lessons Learned

The primary result is the fully functional prototype of the process-driven BIS application, which acts as a use case for evaluating the process engine as well as for the GCM

framework. Appendix C shows the GUI of the BIS application prototype. Additional results have been gained by critically evaluating the application development and experimenting with the application itself. Firstly, it has been successfully deployed on various hardware platforms ranging from one multi-core PC to heterogeneous sets of clusters as provided by the Grid5000 project [116]. Switching hardware platforms did not require changing a single line of functional code, only the infrastructure part of the XML deployment descriptor required modification. The strict separation of concerns and the autonomic functionality implemented within the GCM framework are the main factors leading to this flexibility. The former ensures that resources are never directly referenced in the source code while the latter provides autonomic adaptation to the performance properties of the hardware in use. Also, the lightweight and platform independent nature of the ePVM framework clearly contributed to this.

Secondly, functionality and autonomic behaviour of the application has been verified using Grid5000. The BIS has been started using 50 workers (one per node), a DB holding 50000 identities (approx. 400 MB), and a QoS contract mandating a partition size of 1000 identities/worker. At runtime, the contract has been updated to 800 ($\pm 10\%$) identities/worker. Thereupon, the AM has successfully detected 7 contract violations and each time reconfigured the DP skeleton by adding one additional worker until a partition size of 877 identities/worker was reached at 57 workers/nodes. During this experiment, every reconfiguration operation took about 9 seconds in which the complete DB has been redistributed (from the node hosting the whole database to the nodes hosting the workers of the data parallel BS) by the ABC. When identification requests were issued during reconfiguration, they were queued automatically by the skeleton and processed as soon as reconfiguration was completed. For the given DB size, each identification request required around 10 seconds to be processed. This means that each reconfiguration operation roughly decreases the throughput of the BIS by one identification for any given timeframe. Therefore, if the BIS is used in a very dynamic environment requiring frequent reconfiguration, the number of occurrences of reconfigurations may be sensibly reduced by adopting more aggressive parallelism degree variation policies, in such a way the overall overhead is reduced.

Finally, evaluating the application's source code, including the deployment descriptor required to run on 50 nodes of Grid5000, unveiled the source code breakdown illustrated in Figure 46. The functional code mainly includes the host process objects (c.f. Figure 43) providing DB access, the GUI functionality, and the interfacing to the GCM

components. Its absolute size is about 2500 lines of code, which is very small considering the overall functionality provided by the application. This is due to the fact that the GCM framework provides all the functionality for distribution and autonomicity. Implementing this functionality from scratch not using GCM would have been significantly more effort. In particular, adding autonomic control to an application is virtually effortless if a matching behavioural skeleton is available. Furthermore, it is to be noted that more than a quarter of the source code (27%) consists of code interpreted at runtime. This code, including the deployment descriptor, the process definitions, the QoS contract, and the GCM component definitions, contains the main control logic and infrastructure definition of the application. As a result, the application can be adapted significantly without recompilation - a very important property required for operation in today's dynamic business environments.

Hard-coding this part of the application would clearly decrease the applications flexibility as achieved through the combination of GCM and ePVM.

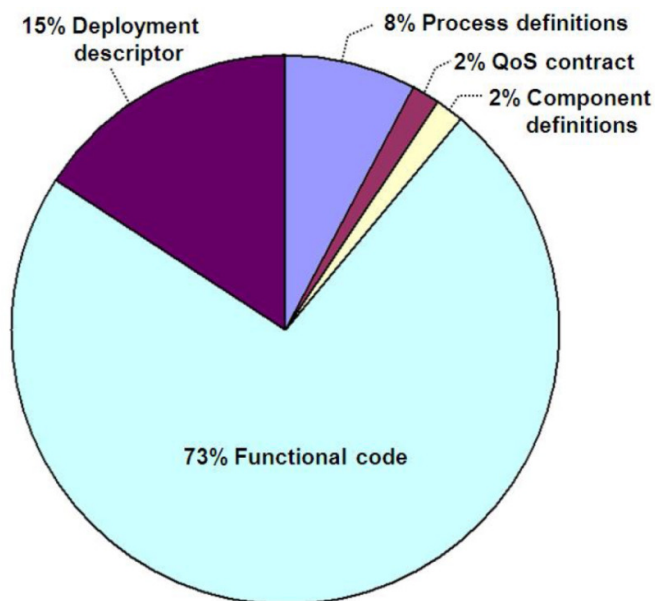


Figure 46. BIS source code breakdown

During application development, we have made a number of experiences with regards to the integration of process technology and the GCM framework. The interfacing between the two technologies went rather smoothly, since the ePVM engine is available as a Java library and it does not dictate the use of Web services. Also, the DP skeleton fits well to the given biometric identification problem. However, application monitoring turned out

to be challenging. One must be aware that the idea behind components is to hide complexity and this can be a problem if component internals need to be monitored. The GCM framework supports querying the state of a component system, however, it does not support monitoring activities within components, for example, reconfiguration within a BS. Solving this problem by instrumenting component implementations requires comprehensive knowledge of the GCM framework. Furthermore, the monitoring support of GCM follows a pull model while process engines are mostly event driven. Joining the two paradigms in a sensible way requires an extra effort and can have a performance impact. For example, regularly traversing component hierarchies to detect newly created components is not very efficient.

Another lesson we have learned is that the two different synchronization paradigms applied in GCM and ePVM can interfere if not handled with care. The concept of automatic futures implemented in the GCM framework follows the wait-by-necessity idea. This means that unavailable results are replaced by future objects such that synchronization is delayed as long as possible. Therefore, it must be carefully checked if results of activities within a process flow include one or more future objects before the next activity of a sequence is triggered, otherwise the process semantics can easily become distorted. In other words, if a GCM component returns an object it does not necessarily mean that all the related operations have completed.

Apart from such specific details to be aware of, the ePVM framework has proven to be a very practical and efficient foundation for building process-driven applications. The few requirements it poses to its environment and the clear and lean API lead to a very smooth integration with GCM. In particular, the approach of state machine programming using a simple scripting language made the continuous modification of process definitions required during the development phase very easy. Additionally, the host process object interface, as the generic interface into the host application environment has proven to be powerful. Together with the communicating state machine model of the ePVM engine, it was possible to define and control concurrency and synchronization, within the process engine as well as on its boundary, without having to deal with threads and basic synchronization primitives directly. Overall, the ePVM framework has proven to be embeddable, versatile, and easily programmable.

Finally, extending the BIS by integrating secure transaction authorization capabilities of the ZTIC has shown that the ZTIC approach is really generic and easy to integrate. It

demonstrates that the ZTIC does not only deliver security but also proves its ease-of-use, mobility, and integration properties in a real-world scenario.

5.4. SUMMARY AND CONTRIBUTIONS

This chapter presents the evaluation of the results of this thesis. In principle, evaluation was carried out in two phases. Firstly, the basic features and properties of the ePVM framework and the ZTIC approach have been evaluated through a series of experiments. This way the general suitability of the architecture as well as certain performance properties could be verified. The results showed that the ePVM architecture is highly suitable as a core framework for building workflow systems and process-driven applications. The programming model and programming interfaces turned out to be efficient and execution performance is good considering the prototype stage of the framework implementation. Furthermore, flexibility and efficiency of the proposed persistence model have been validated via an initial experiment. The properties of the ZTIC approach have been verified successfully through the simulated Internet banking application.

Secondly, the results of this thesis have been further evaluated by using them to build a comprehensive process-driven application – the biometric identification system. Here, the ePVM framework has been embedded in a distributed application making use of autonomic GCM components. The generic nature of the ePVM framework and its versatility could be successfully validated in such a challenging environment. Finally yet importantly, securing the BIS application via the ZTIC supporting multi-party transaction authorization demonstrated the power of combining ePVM with the unique feature set provided by the ZTIC.

This chapter is based on work published in [51,49,52,53,54].

6. CONCLUSIONS AND FUTURE DIRECTIONS

The conclusive chapter of this thesis briefly summarizes the achievements and deliverables of this research project, draws a conclusion about the applicability of the ePVM framework and the ZTIC based on the analysis and evaluation, and discusses possible future directions in pursuit of a complete process framework and an advanced authentication and transaction authorization solution.

6.1. SUMMARY AND ACHIEVEMENTS

Process execution engines are not only an integral part of workflow and business process management systems but are increasingly used to build process-driven applications. In other words, they are potentially used in all kinds of software across all application domains. This approach delivers a number of benefits, for instance, the application logic can be modified without re-compiling the application, the business logic is more evident, and various features of the process engine such as support for process monitoring and process persistence can be exploited. However, contemporary process engines and workflow systems are unsuitable to be used in such diverse application scenarios for several reasons. The main shortcomings can be observed in the areas of interoperability, versatility, and programmability. For instance, many process engines are rather heavyweight, monolithic systems that are built for particular domain specific process definition languages. Additionally, most process languages are XML based declarative languages that require high-level tools such as graphical modelling and code generation tools.

Therefore, this thesis made a step away from domain specific monolithic workflow engines towards generic and versatile process runtime frameworks, which enable the integration of process technology into all kinds of software. As a result, the idea and corresponding architecture of a lightweight and generic process virtual machine (ePVM) has been presented, which supports defining process flows along the theoretical foundation of communicating extended finite state machines. The architecture focuses on core process functionality such as control flow and state management, monitoring, persistence, and communication, while using JavaScript as the process definition language. This approach led to a very generic yet easily programmable process framework. Along with the ePVM architecture, a persistence model for state-machine

based process engines has been presented, which leads to improved flexibility in defining persistence behaviour and thus enables optimization of performance and resource consumption during process execution. A fully functional prototype implementation of the proposed ePVM framework has been developed and an initial example application illustrating the usage of the framework was provided.

Despite the fact that business processes are increasingly automated and controlled by information systems, humans are still involved in many of them, directly or indirectly. People are involved when critical decisions are to be made during a process flow. Thus, for process flows involving sensitive transactions, a highly secure authorization scheme supporting asynchronous multi-party transaction authorization must be available within process management systems. Therefore, along with the ePVM framework, this thesis has presented a novel approach for secure remote multi-party transaction authentication - the zone trusted information channel (ZTIC). The ZTIC approach uniquely combines multiple desirable properties such as the highest level of security, ease-of-use, mobility, remote administration, and smooth integration into existing infrastructures into one device and method. In this thesis, the idea and architecture of the ZTIC has been developed and presented along with a corresponding ePVM framework extension that integrates the ZTIC approach into the framework.

Both, the ePVM framework and the ZTIC, have been extensively evaluated separately as well as in the context of a complex process-driven use case application focusing on large-scale biometric identification. The application, developed as part of the GridCOMP EU project, has been used as a real-world case study for gaining experience and validating the results of this thesis. Finally, the results have been presented and critically discussed. In summary, the main deliverables of this thesis project supporting the hypotheses stated in Section 1.4 are:

- ❑ Detailed analysis and critical evaluation of existing process engines authentication and transaction authorization methods
- ❑ Architecture and prototype implementation of a generic and embeddable process execution framework including a flexible persistence model
- ❑ A novel device and method for secure remote authentication and multi-party transaction authorization
- ❑ Evaluation and validation of the prototypes via a series of experiments and a comprehensive process-driven use case application

6.2. CONCLUSIONS

The main goal of this thesis was to address and provide solutions for the challenges and requirements stated beforehand. The challenges of interoperability, versatility, and programmability of process execution frameworks have been comprehensively addressed by the ePVM framework architecture and its prototype implementation. Furthermore, the definition of the ZTIC clearly addressed the issues of contemporary authentication and transaction authorization methods, namely, the effectiveness against software attacks, ease-of-use, mobility, administration, and integration. In addition, the two project deliverables have been combined to amplify their value.

Having completed extensive evaluation of the deliverables, it can be concluded that ePVM in combination with the ZTIC approach represents a unique and very powerful framework for building workflow systems and process-driven applications including support for secure multi-party transaction authorization. This conclusion is based on the results and analysis presented in this thesis, more specifically:

- the adequacy of the ePVM architectural design validated by experiments and the comprehensive use case application;
- the suitability of the ePVM APIs, the concept of host process objects, and the monitoring interface illustrated through the example applications;
- the efficiency and simplicity of the CEFSM based programming model provided by the ePVM JavaScript runtime environment as illustrated through the example process definitions;
- the lightweight, generic, and versatile nature of the ePVM framework illustrated by the example and use case applications;
- the flexibility and effectiveness of the persistence model implemented in ePVM
- the suitability of the ZTIC approach with respect to security, ease-of-use, mobility, administration, and integration as demonstrated by the example applications;
- the successful integration of the ZTIC approach into the ePVM framework as demonstrated by the Internet bank application and the GridCOMP use case.

6.3. FUTURE DIRECTIONS

The ePVM framework together with the ZTIC, which has been turned into a real prototype by colleagues within the IBM research team, provides a fully functional process execution and transaction authorization platform. However, a number of

interesting issues towards the extension of both prototypes remain open and are subject to future research.

- **Language mappings:** To further emphasize versatility and interoperability, it is desirable to implement two or more mappings of popular domain specific process definition languages to the ePVM model. Obviously, support for BPEL would be one candidate. At the time of writing, it is planned to develop an ePVM extension that maps BPMN process definitions to ePVM. BPMN is increasingly gaining interest in the industry and seems to become a widespread standard for business process modelling. However, BPMN is a graphical notation mainly used for drawing process graphs. A mapping to ePVM would allow turning BPMN diagrams into executable processes.
- **Interface definitions:** At the time of writing, ePVM does not provide a formal way of defining process interfaces. In other words, it is left open how to define the format of the messages a state machine can process. In current applications, JSON formatted comments have been used to document message formats. However, a well-defined format for documenting the messaging interfaces would be desirable.
- **Transaction support:** Based on the presented persistence model, ePVM can be extended to support local transactions. The exact semantics must be defined and a corresponding runtime API that allows beginning, committing, and aborting of transactions must be provided.
- **Process modification:** In some application scenarios, it can make sense to modify a process definition of certain process instances at runtime. Effectively, this means that the process function is modified while the process instance exists already. Support for such modifications should be added to ePVM.
- **Preemptive processes:** The general rule within the ePVM model is that message processing of process instances is always short lived. This means that computational intensive code should be loaded off to host process objects such that the code is executed by dedicated OS threads. However, there might be situations where it makes sense to do such processing in JavaScript, for instance, if it is easier to implement or if off-loading would require a lot of communication. For such cases, process instances of type PREEMTIVE could be introduced. Preemptive processes would then be executed by OS threads instead of ePVM

threads with the trade-off that they would not be able to share memory or communicate with other process instances, as they would run in their own JavaScript context.

- **Smart card based ZTIC:** There is the idea to produce new ZTIC hardware that uses a standard smart card chip as the main controller. This would increase physical security of the device, as smart card chips are the most secure one-chip solutions available. Additionally, it would reduce device cost. For implementing the second channel approach, a standard smart card chip should be powerful enough. Also, moving from USB MSD to a USB human interface device (HID) is being considered, as it would further increase compatibility in cases where MSD support is not available.
- **Social attacks:** Social attacks are one of the attack vectors not addressed by the ZTIC approach. The assumption is that the user is sufficiently trained to rely on the ZTIC display and reject any transaction that appears questionable. At the time of writing, it is unclear how hard it is to successfully train users or how many users can be convinced to approve a fraudulent transaction request via a social attack. Also, there is no approach that successfully thwarts social attacks. Thus, more research is required into that direction, as in practice such attacks can be quite easily launched.

REFERENCES

- [1] C. A. Ellis and G.J. Nutt, "Office Information Systems and Computer Science," *ACM Computing Surveys*, vol. 12, no. 1, pp. 27-60, 1980.
- [2] D. E. Mahling, N. Craven, and W. B. Croft, "From Office Automation to Intelligent Workflow Systems," *IEEE Expert*, no. 10, pp. 41-47, June 1995.
- [3] Rob Allen, "Workflow: An Introduction," in *Workflow Handbook.: Future Strategies*, 2001, pp. 15-38.
- [4] Michael zur Muehlen, *Workflow-based Process Controlling - Foundation, Design, and Application of Workflow-driven Process Information Systems*. Berlin: Logos Verlag, 2004.
- [5] David Hollingsworth, "The Workflow Reference Model," Workflow Management Coalition, Winchester, Document Number TC00-1003, 1995.
- [6] David S. Linthicum, *Enterprise Application Integration.:* Addison-Wesley, 1999.
- [7] Steve Graham et al., *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*, 2nd ed.: Sams, 2004.
- [8] Eric Newcomer and Greg Lomow, *Understanding SOA with Web Services.:* Addison Wesley, 2005.
- [9] Howard Smith and Peter Fingar, *Business Process Management (BPM): The Third Wave*, 2nd ed.: Meghan Kiffer, 2006.
- [10] Paul Harmon, *Business Process Change, Second Edition: A Guide for Business Managers and BPM and Six Sigma Professionals*, 2nd ed., Morgan Kaufmann, Ed., 2007.
- [11] A. Iyenga, V. Jessani, and Chilanti M., *WebSphere Business Integration Primer: Process Server, BPEL, SCA, and SOA*, 1st ed.: IBM Press, 2007.
- [12] Matt Cumberlidge, *Business Process Management with JBoss jBPM: A Practical Guide for Business Analysts.:* Packt Publishing, 2007.
- [13] B. Halipern and P. Tarr, "Model-driven development: The good, the bad, and the ugly," *IBM Systems Journal*, vol. 45, no. 3, pp. 451-461, 2006.
- [14] S. A. White, D. Miers, and L. Fischer, *BPMN Modeling and Reference Guide*, 1st ed.: Future Strategies Inc., 2008.

- [15] Group Object Management (OMG). (2005, July) UML Version 2.0. [Online]. <http://www.omg.org/spec/UML/2.0/>
- [16] Organization for the Advancement of Structured Information Standards (OASIS). (2007, April) Web Services Business Process Execution Language (WS-BPEL) Version 2.0. [Online]. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>
- [17] Workflow Management Coalition (WfMC). (2008, October) XML Process Definition Language (XPDL) 2.1. [Online]. <http://wfmc.org/xpdl.html>
- [18] D. Rossi and E. Turrini, "Using a process modeling language for the design and implementation of process-driven applications," in *International Conference on Software Engineering Advances (ICSEA)*, 2007, p. 55.
- [19] Bruce Bukovics, *PRO WF: Windows Workflow in .NET 3.0*, 1st ed.: Apress, 2007.
- [20] T. Weigold, T. Kramp, and M. Baentsch, "Remote Client Authentication," *IEEE Security & Privacy Journal*, vol. 6, no. 4, pp. 36-43, July-August 2008.
- [21] A. Hiltgen, T. Kramp, and T. Weigold, "Secure Internet Banking Authentication," *IEEE Security & Privacy Journal*, vol. 4, no. 2, pp. 21-29, March-April 2006.
- [22] R. Dhamija et al., "Why Phishing Works," in *Human Factors in Computing Systems*, 2006, pp. 581-590.
- [23] B. Schneier, "Two-Factor Authentication: Too Little, Too Late," *Communications of the ACM*, vol. 48, no. 4, p. 136, 2005.
- [24] Binational Working Group on Cross-Border Mass Marketing Fraud. (2006, October) US Department of Justice & Ministry on Public Safety. [Online]. www.usdoj.gov/opa/report_on_phishing.pdf
- [25] Swiss Federal Office of Police. (2007) Semi-annual Report, Swiss Reporting and Analysis Centre for Information Assurance (MELANI). [Online]. <http://www.melani.admin.ch/dokumentation/00123/00124/01029/index.html?lang=en>
- [26] AXSionics AG. (2009) The Internet Passport. [Online]. <http://www.axsionics.ch/>
- [27] Federal Office for Information Security. (2007) The IT Security Situation in Germany 2007. [Online]. http://www.bsi.de/english/publications/securitysituation/Lagebericht_2007_englisc

h.pdf

- [28] Zentraler Kreditausschuss Germany. (2009) Secoder standard for chipcard reader. [Online]. <http://www.zka-online.de/zka/zahlungsverkehr/secoder-1.html>
- [29] Y. Byun, Sanders B., and Keum C., "Design Patterns of Communicating Extended Finite State Machines in SDL," in *Conference on Pattern Languages of Programs (PLoP)*, Monticello, 2001.
- [30] Mozilla.org. Rhino: JavaScript for Java. [Online]. <http://www.mozilla.org/rhino/>
- [31] GridCOMP EU project, "Grid Programming with Components, An Advanced Component Platform for an Effective Invisible Grid," <http://gridcomp.ercim.org>, 2009.
- [32] Trevor Naidoo and Michael zur Muehlen. (2005) The State of Standards and their Practical Application, AIIM Conference 2005. [Online]. [http://www.workflow-research.de/Publications/Slides/TRNA.MIZU-State_of_Standards_\(AIIM_2005-05-17\).pdf](http://www.workflow-research.de/Publications/Slides/TRNA.MIZU-State_of_Standards_(AIIM_2005-05-17).pdf)
- [33] Michael zur Muehlen, "Tutorial - Business process management standards," in *Proceedings of the 5th International Conference on Business Process Management (BPM '07)*, 2007.
- [34] Ryan K. L. Ko, "A Computer Scientist's Introductory Guide to Business Process Management (BPM)," *ACM Crossroads*, vol. 15, no. 4, pp. 11-18, 2009.
- [35] N. Russell, A. Hofstede, van der Aalst, and W.M.P., "newYAWL: Specifying a Workflow Reference Language using Coloured Petri Nets," in *Eighth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, Aarhus, 2007, pp. 107-127.
- [36] Tom Baeyens. (2008, February) Process Component Models: The Next Generation in Workflows?, InfoQ article. [Online]. <http://www.infoq.com/articles/process-component-models>
- [37] Tom Baeyens and M. V. Faura. (2007, September) The Process Virtual Machine, OnJava article. [Online]. <http://www.onjava.com/pub/a/onjava/2007/05/07/the-process-virtual-machine.html>
- [38] S. M. Fernandes, J. Cachopo, and A. R. Silva, "Supporting Evolution in Workflow Definition Languages," in *SOFSEM 2004*, 2004, pp. 49-59.

- [39] Dragos A. Manolescu, "An Extensible Workflow Architecture with Objects and Patterns," in *Technology of Object-Oriented Languages, Systems, and Architectures*.: Kluwer Academic Publishers, 2003, ch. 4.
- [40] Paul Andrew et al., *Presenting Windows Workflow Foundation*, 1st ed.: Sams, 2005.
- [41] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, and D. Roller, "Business process choreography in WebSphere: Combining the," *IBM Systems Journal*, vol. 43, no. 2, 2004.
- [42] Michael Blow et al. (2004) BPELJ: BPEL for Java. A Joint White Paper by BEA and IBM. [Online]. <http://www.ibm.com/developerworks/library/specification/ws-bpelj/>
- [43] JBoss. (2009) JBoss jBPM. [Online]. <http://www.jboss.com/products/jbpm/>
- [44] JBoss. (2009) JBoss jBPM User Guide. [Online]. <http://docs.jboss.org/jbpm/v3/userguide/>
- [45] Zentraler Kreditausschuss. (2009) Zahlungsverkehr mit Chipkartenleser Secoder. [Online]. <http://www.zka-online.de/zka/zahlungsverkehr/secoder-1.html>
- [46] European Committee for Standardization (CEN). (2009) Financial transactional IC card reader (Finread) part 1-8. [Online]. <http://www.cen.eu/cenorm/sectors/sectors/iss/cen+workshop+agreements/finread.asp>
- [47] Barclays. (2009) Introducing PINsentry for Online Banking. [Online]. <http://www.barclays.co.uk/pinsentry/>
- [48] T. Kramp, P. Buhler T. Weigold, "ePVM – An Embeddable Process Virtual Machine," in *Proceedings of the 31st Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, Beijing, China, 2007, pp. 557-564.
- [49] T. Weigold, T. Kramp, and P. Buhler, "Flexible Persistence Support for State Machine-based Workflow Engines," in *Proceedings of the 4th IEEE International Conference on Software Engineering Advances (ICSEA)*, Porto, 2009, pp. 313-319.
- [50] T. Weigold et al., "The Zurich Trusted Information Channel - An Efficient Defense Against Man-in-the-middle and Malicious Software Attacks," in *Proceedings of TRUST 2008*, 2008, pp. 75-91.

- [51] T. Weigold, M. Aldinucci, M. Danelutto, and V. Getov, "Integrating Autonomic Grid Components and Process-Driven Business Applications," in *Proceedings of the 3rd International ICST Conference on Autonomic Computing and Communication Systems (Autonomics)*, Limassol, 2009.
- [52] T. Weigold, P. Buhler, A. Basukoski, and V. Getov, "Advanced Grid Programming with Components: A Biometric Identification Case Study," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, Turku, Finland, 2008, pp. 401-408.
- [53] P. Buhler, V. Getov, S. Isaiadis, T. Weigold A. Basukoski, "Methodology for Component-based Development of Grid Applications," in *Proceedings of CompFrame/HPC-GECO workshop on Component Based High Performance Computing (CBHPC)*, ACM Digital Library, ISBN:978-1-60558-311-2, Karlsruhe, Germany, 2008.
- [54] T. Weigold, M. Aldinucci, M. Danelutto, and V. Getov, "Development Methodology for Autonomic Process-Driven Business Applications," *Int. Journal of Autonomous and Adaptive Communications systems (IJAACS)*, (to appear) 2010.
- [55] K.P. Stoilova and T.A. Stoilov, "Evolution of the workflow management systems," in *XLI International Scientific Conference on Information, Communication and Energy Systems and Technologies*, Sofia, 2006, pp. 225-228.
- [56] IBM. (2009, June) WebSphere Process Server for Multiplatforms. [Online]. ftp://ftp.software.ibm.com/software/websphere/integration/wps/library/pdf620/overview_wpsmp_en.pdf
- [57] Thomas Volmering, "Business Process Management mit SAP NetWeaver und ARIS — Eine gemeinsame Sprache für Business und IT ," in *Von Prozessmodellen zu lauffähigen Anwendungen*. Berlin: Springer, 2005, pp. 35-47.
- [58] Jeff Davies et al., *The Definitive Guide to SOA: BEA AquaLogic Service Bus.*: Apress, 2007.
- [59] Open Service Oriented Architecture (OSOA). (2009) Service Component Architecture (SCA) Specifications. [Online]. <http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications>
- [60] Dragos Manolescu, "MICRO-WORKFLOW: A WORKFLOW ARCHITECTURE

- SUPPORTING COMPOSITIONAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT," University of Illinois, PhD thesis 2000.
- [61] Peter Muth et al., "Integrating light-weight workflow management systems within existing business environments," in *15th International Conference on Data Engineering*, Sydney, 1999, p. 286.
- [62] Claus Johannes Hagen, "A Generic Kernel for Reliable Process Support," Swiss Federal Institute of Technology, Zuerich, PhD Thesis 1999.
- [63] Björn Axenath, Ekkart Kindler, and Vladimir Rubin, "An Open and Formalism Independent Meta-Model for Business Processes," in *Workshop on Business Process Reference Models 2005 (BPRM 2005)*, Nancy, 2005, pp. 45-59.
- [64] The Enhydra.org Initiative. (2009, July) Enhydra Shark Open Source Java XPDL Workflow. [Online]. <http://www.enhydra.org/workflow/shark/index.html>
- [65] active endpoints. (2009, July) The ActiveBPEL Engine. [Online]. <http://www.activevos.com/community-open-source.php>
- [66] Steve Freeman and Nat Pryce, "Evolving an Embedded Domain-Specific Language in Java," in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, 2006, pp. 855-865.
- [67] James Elliott, Tim O'Brien, and Ryan Fowler, *Harnessing Hibernate*, 1st ed.: O'Reilly, 2008.
- [68] C. A. Petri, "Kommunikation mit Automaten," Fakultät fuer Mathematik und Physik, Technische Hochschule Darmstadt, PhD thesis 1962.
- [69] R. Milner, "Communicating and Mobile Systems: The Pi-Calculus," Cambridge University, Cambridge, 1999.
- [70] Gang Xue, Joan Lu, and Shaowen Yao, "Investigating Workflow Patterns in Term of Pi-calculus," in *11th International Conference on Computer Supported Cooperative Work in Design*, 2007, pp. 823-827.
- [71] Wil van der Aalst, "Pi Calculus Versus Petri Nets - Let us eat "humble pie" rather than further inflate the "Pi hype"," *BPTrends*, pp. 1-11, May 2005.
- [72] Fei Xu and Li Zhang, "nified Modeling and Analysis based on Petri nets and Pi calculus," in *1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE '07)*, 2007, pp. 75-86.

- [73] W.M.P. van der Aalst et al., "Correctness-Preserving Configuration of Business Process Models," in *11th International Conference on Fundamental Approaches to Software Engineering (FASE 2008)*, 2008, pp. 46-61.
- [74] W.M.P. van der Aalst and A.H.M. ter Hofstede, "YAWL: Yet Another Workflow Language," Queensland University of Technology, Brisbane, QUT Technical report FIT-TR-2003-04, 2003.
- [75] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5-51, July 2003.
- [76] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. der Hofstede, "Design and implementation of the YAWL system," in *16th International Conference on Advanced Information Systems Engineering (CAiSE)*, Riga, 2004, pp. 142-159.
- [77] N. Russell and W.M.P. van der Aalst, "newYAWL: Designing a Workflow Systems Using Coloured Petri Nets," in *International Workshop on Petri Nets and Distributed Systems*, 2008, pp. 67-84.
- [78] BigBross Open Source Software. (2009, July) Bossa Workflow System. [Online]. <http://www.bigbross.com/bossa/>
- [79] Humboldt University Berlin. (2009, July) Petri Net Markup Language. [Online]. <http://www2.informatik.hu-berlin.de/top/pnml/about.html>
- [80] OpenSymphony Quality Components. (2009, July) OSWorkflow. [Online]. <http://www.opensymphony.com/osworkflow/>
- [81] BeanShell developer group. (2009, July) BeanShell - Lightweight Scripting for Java. [Online]. <http://www.beanshell.org/>
- [82] L. Lamport, "Password Authentication with Insecure Communication," *Communications of the ACM*, vol. 24, no. 11, pp. 770-772, 1981.
- [83] R.E. Smith, *Authentication: From Passwords to Public Keys.*: Addison-Wesley, 2002.
- [84] U. Waldmann, "Protected Transmission of Biometric User Authentication Data for On-card-Matching," in *Proceedings of the 2004 ACM symposium on Applied computing*, Nicosia, 2004, pp. 425-430.
- [85] X. Leroy, "Java Bytecode Verification: Algorithms and Formalizations," *Journal*

- of Automated Reasoning*, vol. 30, pp. 235-269, 2003.
- [86] NFC Forum. (2009, July) NFC Forum Website. [Online]. <http://www.nfc-forum.org/home>
- [87] ISO/IEC Working Group 8 (WG8). (2009, July) ISO/IEC 14443 Proximity Cards. [Online]. <http://wg8.de/sd1.html#14443>
- [88] 3GPP. (2009, July) 3GPP Specifications Download. [Online]. www.3gpp.org/ftp/Specs/archive/11_series/11.14/1114-850.zip
- [89] Michael Steiner et al., "Secure Password-Based Cipher Suite for TLS," *ACM Transactions*, vol. 4, no. 2, pp. 134-157, 2001.
- [90] K. Fu et al., "Dos and Don'ts of Client Authentication on the Web," in *Proceedings of the Usenix Security Forum*, 2001, pp. 251-268.
- [91] Crealogix. (2009, August) CLX.Sentinel e-Banking security. [Online]. http://www.crealogix.ch/fileadmin/Leistungsangebot/pdf/products/e-banking/factsheet-clx_sentinel.pdf
- [92] Kobil. (2009, August) mIDentity. [Online]. <http://www.kobil.com/de/produkte/midentity-technologie/midentity.html>
- [93] W.M.P. van der Aalst, "The Application of Petri Nets to Workflow Management," *The Journal of Circuits, Systems and Computers*, vol. 8, no. 1, pp. 21-66, 1998.
- [94] P. A. Sheth, W. van der Aalst, and I. B. Arpinar, "Processes Driving the Networked Economy," *IEEE Concurrency*, vol. 7, no. 3, pp. 18-31, July 1999.
- [95] N. Russell et al., "Workflow Data Patterns," Queensland University of Technology, QUT Technical Report FIT-TR-2004-01, 2004.
- [96] Rik Eshuis and Juliane Dehnert, "Reactive Petri Nets for Workflow Modeling," in *Proceedings of 24th International Conference on Applications and Theory of Petri Nets (ICATPN)*, Eindhoven, 2003, pp. 296-315.
- [97] Daniel Brand and Pitro Zafiropulo, "On Communicating Finite-State Machines," *Journal of the ACM*, vol. 30, no. 2, pp. 323-342, April 1983.
- [98] Thorsten Kramp, *Flexible Run-Time Support for Real-Time Computing*. Aachen: Shaker Verlag, 2001.
- [99] M.Z. Beiroumi and V.B. Iversen, "Recovery method based on communicating extended finite state machine (CEFSM) for mobile communications," in

- Proceedings of 10th International Conference of Engineering of Complex Computer Systems (ICECCS)*, 2005, pp. 384 - 393.
- [100] Richard Ekwall et al., "Towards Flexible Finite-State-Machine-Based Protocol Composition," in *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications*, 2004, pp. 281-286.
- [101] Miro Samek, *Practical Statecharts in C/C++*. Lawrence, Kansas: CMP Books, 2002.
- [102] International Telecommunication Union (ITU). (2009, August) Specification and Definition Language (SDL), ITU-T Recommendation Z.100. [Online]. http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf
- [103] European Computer Manufacturers Association (ECMA). (2009, August) ECMAScript Language Specification, Standard ECMA-262, 3rd Edition, December 1999. [Online]. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf>
- [104] European Computer Manufacturers Association (ECMA). (2009, August) ECMAScript for XML (E4X) Specification, Standard ECMA-357. [Online]. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-357.pdf>
- [105] H. Schuldt, G. Alonso, C. Beerli, and H. Schek, "Atomicity and isolation for transactional processes," *ACM Transactional Database Systems*, vol. 27, no. 1, pp. 63-116, 2002.
- [106] D. Crockford. (2009, September) JavaScript Object Notation (JSON), RFC 4627. [Online]. <http://www.json.org/>
- [107] M. Baentsch et al., "A Banking Server's Display on Your Key Chain," *ERCIM News*, no. 73, pp. 44-45, 20008.
- [108] (2009, October) The Uniform Server. [Online]. <http://www.uniformserver.com/>
- [109] CoreGrid NoE. (2007) CoreGRID NoE deliverable series, Institute on Programming Model: Deliverable D.PM.04 - Basic Features of the Grid Component Model (assessed). [Online]. <http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf>
- [110] Object Web Consortium. (2003) The Fractal Component Model, Technical Specification. [Online]. <http://fractal.ow2.org/>

- [111] R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., Smolinski, B. Armstrong, "Toward a common component architecture for high performance scientific computing," in *Proc. of the 8th Intl. Symposium on High Performance Distributed Computing (HPDC)*, 1999, p. 13.
- [112] M. Cole, "Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, no. 30, pp. 389-406, 2004.
- [113] M., Campa, S., Danelutto, M., Dazzi, P., Kilpatrick, P., Laforenza, D., Tonellotto, N. Aldinucci, "Behavioural skeletons for component autonomic management on grids," in *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, Heraklion, Greece, 2007.
- [114] M., Danelutto, M., Kilpatrick, P Aldinucci, "Autonomic management of non-functional concerns in distributed and parallel application programming," in *Proceedings of Intl. Parallel & Distributed Processing Symposium (IPDPS)*, Rome, Italy, 2009, pp. 1-12.
- [115] L. Henrio, L. Cardelli D. Caromel, *A Theory of Distributed Objects.*: Springer Verlag, ISBN:3540208666 , 2004.
- [116] The Grid5000 Project. (2009) An infrastructure distributed in 9 sites around France, for research in large-scale parallel and distributed systems. [Online]. <https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home>

Appendix A. EPVM HOST AND RUNTIME API

As the full API specification would significantly increase the size of this thesis document, we decided not to include it completely. However, to give an idea how the main functionalities of the host and runtime programming interfaces are provide to application programmers, this appendix includes the method summary of the two most important classes. It includes the main methods a developer works with when embedding ePVM or when developing process definitions, respectively.

A.1. EPVM HOST API

Below is the method summary of the Java class *ProcessVM*. This is the main class used for embedding ePVM into Java applications.

Method Summary	
PVMObject	buildObject (java.lang.Object data) Creates a PVMObject representing a JavaScript Object, Array, or XML object.
java.lang.Object	call (java.lang.String targetProcess, java.lang.Object message) Sends a message to a process instance synchronously.
java.lang.String	deploy (java.lang.String processPackage, java.lang.String sourceInfo, java.io.PrintWriter out) Deploys a process package to the ePVM engine.
void	forward (java.lang.String targetProcess) This method can be used by a host process to forward the current message unmodified.
void	forward (java.lang.String targetProcess, java.lang.Object message) This method can be used by a host process to forward the current message with modified message value.
static java.lang.String	getVersion () Returns ePVM version information.
java.lang.Object[]	info (java.lang.String handle) Provides info about a given process.
java.lang.String	invoke (java.lang.String packageName, java.lang.String processName) Invokes a process by creating a new process instance.
java.lang.String	invoke (java.lang.String packageName, java.lang.String processName, int type, PVMObject environment, int priority) Invokes a process by creating a new process instance.
java.lang.String	join (java.lang.String targetProcess, java.lang.String packageName, java.lang.String processName, PVMObject environment) Creates a process instance that joins the given process instance to form a process group.
java.lang.String[]	packageList () Lists currently deployed process packages.
java.lang.String[]	processList (java.lang.String packageName) Lists all process instances of a given package or all registered host processes.

java.lang.String	registerHostProcess (FVMHostProcess hostProcess, java.lang.String processName) Register a host process object with the ePVM engine.
void	registerMonitor (FVMMonitor monitor, java.lang.String packageName, int events, boolean synchronous) Registers an event monitor for a process package with the ePVM runtime.
void	reply () This method can be used by host processes to send a reply to the current message while the message value remains unchanged.
void	reply (java.lang.Object message) This method can be used by host processes to send a reply to the current message.
void	send (java.lang.String targetProcess, java.lang.Object message) Sends a message to a process instance asynchronously.
void	setLogger (java.io.PrintWriter logger, int logLevel) Set the output stream the ePVM engine writes log messages to.
void	start () Starts the ePVM execution engine.
void	stop () Stops the ePVM execution engine.
void	undeploy (java.lang.String packageName) Undeploy a process package from the ePVM engine.
void	unregisterHostProcess (java.lang.String processName) Unregister a host process currently registered with the ePVM engine.
void	unregisterMonitor (java.lang.String packageName) Unregisters the monitor of a process package, if any.

A.2. EPVM RUNTIME API

Below is the method summary of the JavaScript runtime API provided by ePVM. As there is no standard documentation system available for JavaScript, the functions have been documented as Java methods of the dummy class *PVMSystem* using JavaDoc.

Method Summary	
java.lang.Object	PVM call (java.lang.Number targetProcess, java.lang.Object messageValue) Calls another process.
void	PVM forward (java.lang.Number targetProcess, java.lang.Object messageValue) Forwards the current message.
void	PVM forwardSaved (java.lang.Number targetProcess, java.lang.Object messageValue, java.lang.String sender, java.lang.Object id, java.lang.Object value) Forwards a message from the save-queue.
java.lang.Number	PVM hostProcess (java.lang.String name) Returns the handle of the host process with the given name.
java.lang.Object	PVM info (java.lang.Number handle) Provides info about a given process.

java.lang.Number	PVM invoke (java.lang.Object processDefinition, java.lang.Number type, java.lang.Object environment, java.lang.Number priority) Invokes a process by creating a process instance.
java.lang.Number	PVM invoke (java.lang.String packageName, java.lang.String processName, java.lang.Number type, java.lang.Object environment, java.lang.Number priority) Invokes a process by creating a process instance.
java.lang.Number	PVM join (java.lang.Number targetProcess, java.lang.Object processDefinition, java.lang.Object environment) Creates a process instance that joins the given process instance to form a process group.
java.lang.Number	PVM join (java.lang.Number targetProcess, java.lang.String packageName, java.lang.String processName, java.lang.Object environment) Creates a process instance that joins the given process instance to form a process group.
void	PVM monitor (java.lang.Object info) Sends a custom monitor event to the package monitor.
void	PVM println (java.lang.String s) Prints the given string via the <code>PrintWriter</code> registered during process package deployment.
void	PVM purge () Purge save-queue.
java.lang.Object	PVM receive (java.lang.Number mode) Receives a new message.
void	PVM reply (java.lang.Object messageValue) Sends a reply to the current message.
void	PVM replySaved (java.lang.Object messageValue, java.lang.String sender, java.lang.Object id, java.lang.Object value) Sends a reply to a message in the save-queue.
java.lang.Object	PVM retrieve (boolean retrieveSender, boolean remove, java.lang.String sender, java.lang.Object id, java.lang.Object value) Retrieves data from the save-queue.
void	PVM save (java.lang.Object messageValue) Saves the current message in the save-queue.
java.lang.Number	PVM self () Retrieve own process handle.
void	PVM send (java.lang.Number targetProcess, java.lang.Object messageValue) Sends a message to a process asynchronously.
java.lang.Number	PVM sender () Returns the sender of the current message.
void	PVM sort (java.lang.Object compareFunction) Define a sort function and sort the save-queue.
void	PVM yield () Temporarily pause the process and yield control.

Appendix B. BIS INITIALIZATION PROCESS DEFINITION

The ePVM code below represents an excerpt from the BIS-management process package, which defines the BIS initialization process flow as illustrated in Figure 44.

```
__PVMPackage = "BIS-management";

// host processes
grid = null;
db = null;
numIdentities = 0;

// msg:
//   descriptor      - path and file name of the ProActive XML
//                     descriptor file to be deployed
//   DB-path          - directory path to DB file
//   max-partition    - max. partition size
//   DB-size           - desired size of sample DB to be generated
//   algorithm         - desired matching algorithm
//   task-size        - number of matches per task to be submitted to
//                     DP skeleton
//   number-nodes     - number workers to start with
//   reply            - null if OK, error string otherwise
function startup__PVMPProcess(env, msg) {
  var r;

  try {
    if (msg instanceof Error)
      throw msg;

    if (env.state == null) { // initial state
      // get host process handles
      if ( (grid = PVM_hostProcess("GridCompProcess")) == null ||
          (db = PVM_hostProcess("DBProcess")) == null)
        throw new Error("Missing host process.");

      // start nodes and connect to DB (concurrently)
      PVM_send(PVM_invoke(startNodes), msg.descriptor);
      PVM_send(PVM_invoke(connectToDB__PVMPProcess),
        {dbPath:msg["DB-path"], dbSize:msg["DB-size"],
         algorithm:msg.algorithm});
      env.maxPartition = msg["max-partition"];
      env.taskSize = msg["task-size"];
      env.numberNodes = msg["number-nodes"];
      env.dbSize = msg["DB-size"];
      PVM_save();
      env.state = 1;
      return true;
    }
    else if (env.state == 1) {
      env.state++;
      return true;
    }
    else if (env.state == 2) { // Grid and DB ready
      // deploy matching components
      PVM_monitor("deploying GCM components");
      r = PVM_call(grid, {id:3, numberNodes:env.numberNodes});
      if (typeof r == "string") { // error string
        PVM_replySaved(r);
      }
    }
  }
}
```

```

        return false;
    }

    // submit qos contract to farm
    PVM_monitor("submit QoS contract (partition size: " +
        env.maxPartition + ")");
    r = PVM_call(grid, {id:4, matches:env.maxPartition});
    if (typeof r == "string") { // error string
        PVM_replySaved(r);
        return false;
    }

    // distribute DB
    PVM_monitor("distribute DB of " + env.dbSize +
        " identities to " + env.numberNodes +
        " workers");
    r = PVM_call(grid, {id:7, dbSize:env.dbSize,
        taskSize:env.taskSize});
    if (typeof r == "string") { // error string
        PVM_replySaved(r);
        return false;
    }
}

PVM_monitor("BIS initialization successful");

PVM_replySaved(null); // startup successful
return false; // terminate
}
catch (e) {
    if (PVM_retrieve() != null)
        PVM_replySaved("Unexpected exception in startup process: " +
            e);
    else
        PVM_reply("Unexpected exception in startup process: " + e);
    return false; // terminate
}
}

function startNodes(env, descriptorFile) {
    var r;

    // deploy descriptor
    PVM_monitor("starting nodes");
    r = PVM_call(grid, {id:1, pad:descriptorFile});
    if (typeof r == "string") { // error string
        PVM_reply(new Error(r));
        return false;
    }
    PVM_monitor(r + " cores available in the Grid");
    PVM_reply("nodes started");
    return false;
}

// sets numIdentities
function connectToDB__PVMProcess(env, msg) {
    var r;

    PVM_monitor("connecting to database");
    r = PVM_call(db, {id:1, dbdir:msg.dbPath});

```

```
ERROR_STRING: {
    if (r != msg.dbSize) { // error or DB is not of desired size
        if (typeof r == "string" && r.search("No database") < 0)
            break ERROR_STRING;

        PVM_monitor("generating sample DB");
        // generate sample DB using finger.tif
        r = PVM_call(db, {id:3, dbdir:msg.dbPath, image:"finger.tif",
                        dbsize:msg.dbSize, algorithm:msg.algorithm});
        if (typeof r == "string") // error string
            break ERROR_STRING;

        r = PVM_call(db, {id:1, dbdir:msg.dbPath}); // now connect
        if (typeof r == "string") // error string
            break ERROR_STRING;
    }

    numIdentities = r;
    PVM_reply("DB connected");
    return false;
}

PVM_reply(new Error(r));
return false;
}
...
```


Appendix C. BIS APPLICATION GUI

Biometric Identification System

IBM Biometric Identification System GridCOMP Effective Components for the Grids

Person to be Identified

First Name: -----
Last Name: -----
Address: -----

 DB Size: 8000
Time : -----

Matching Identity

First Name: -----
Last Name: -----
Address: -----

Autonomic Manager

Cores/Workers Used

4

QoS Contract

Target partition size (±10%) : 2000
Current partition size : 2000

Identify:

Target partition size: