AALTO UNIVERSITY
School of Science and Technology
Faculty of Information and Natural Sciences
Degree Programme of Computer Science and Engineering

Eemeli Kantola

Synchronizing data between a social networking service and an RDF store via publish/subscribe

Master's Thesis
Helsinki, 3 June 2010

Supervisor:	Professor Heikki Saikkonen, Aalto University
Instructor:	Mikko Vihonen, M.Sc.(Tech.), Futurice Ltd.

AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY

ABSTRACT OF THE
MASTER'S THESIS

Author:	Eemeli Kantola		
Thesis title:	Synchronizing data between a social networking service and an RDF store via publish/subscribe		
Date:	3 June 2010	**Number of pages:**	68
Faculty:	Information and Natural Sciences		
Degree Programme:	Computer Science and Engineering		
Laboratory:	Laboratory of Software Technology		
Supervisor:	Prof. Heikki Saikkonen		
Instructor:	Mikko Vihonen, M.Sc.(Tech.)		

This Thesis presents a publish/subscribe mediator application for synchronizing data between an RDF-based personal Smart Space, provided by Nokia Smart-M3, and Aalto Social Interface (ASI), a Google OpenSocial inspired RESTful Web Service.

As useful information is scattered over a multitude of different internet resources, aggregating data to enable accessing them through a single interface becomes essential. The Semantic Web technologies provide a dynamically extensible platform for building composite services. However, currently there is a lack of necessary tools to enable sharing data between semantic databases and traditional Web Services in practice.

This problem was approached by first conducting a literature study about the current technologies. The results were used in implementing a synchronization agent between Smart-M3 and ASI, which provides existing user data. Python was chosen as the programming language for its flexibility and the provided Smart-M3 Python knowledge processor library. Functions for mapping ASI's hierarchical ontology and data to an RDF graph were written, and both sides were connected with the mediator agent application. Finally, the application's functionality and performance were evaluated.

The main result of this Thesis is a fault-tolerant agent software for synchronizing between Smart-M3 and ASI, and an evaluation of the software and its future possibilities. The agent can be used as such, and it also provides a basis for implementing further agents that connect other Aalto services to Smart-M3 or some other semantic database. This will facilitate wider adoption of the personal Smart Space concept as a framework for intelligently and non-intrusively sharing data between existing internet resources.

Keywords: Smart Spaces, RDF, RDF store, Smart-M3, ASI, REST, social networks

AALTO-YLIOPISTO
TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Eemeli Kantola		
Työn nimi:	Tiedon synkronointi sosiaalisen verkostopalvelun ja RDF-varaston välillä käyttäen julkaisu/tilaus -viestintää		
Päivämäärä:	3.6.2010	**Sivuja:**	68
Tiedekunta:	Informaatio ja luonnontieteet		
Koulutusohjelma:	Tietotekniikan koulutusohjelma		
Laboratorio:	Ohjelmistotekniikan laboratorio		
Työn valvoja:	Prof. Heikki Saikkonen		
Työn ohjaaja:	DI Mikko Vihonen		

Tämä diplomityö esittelee julkaisu/tilaus -välittäjäsovelluksen tiedon synkronointiin Nokia Smart-M3:n, RDF-pohjaisen henkilökohtaisen Smart Space:n ja Aalto Social Interfacen (ASI), Googlen OpenSocial -tyylisen REST-pohjaisen web servicen välillä.

Hyödyllinen tieto on usein hajallaan monen eri internet-resurssin takana, jolloin tiedon yhteen kokoaminen ja tarjoaminen yhteisen rajapinnan kautta on oleellista. Semanttisen webin teknologiat tarjoavat dynaamisesti laajennettavan alustan yhdisteltyjen palvelujen tarjoamiseen. Tällä hetkellä kuitenkin tarvittavien työkalujen puute hankaloittaa tiedon jakamista semanttisten tietokantojen ja perinteisten web-palvelujen välillä.

Tätä ongelmaa lähestyttiin ensin kirjallisuustutkimuksella nykyisistä teknologioista. Tuloksia hyödynnettiin synkronointiagentin toteuttamisessa Smart-M3:n ja ASI:n välille, joista jälkimmäinen sisältää olemassaolevaa käyttäjädataa. Python valittiin ohjelmointikieleksi sen monipuolisuuden ja Smart-M3:n tarjoaman kirjastotuen takia. Funktiot kirjoitettiin ASI:n hierarkkisen ontologian muuntamiseen RDF-verkoksi, ja molemmat osapuolet yhdistettiin välittäjäagenttisovelluksella. Lopuksi sovelluksen toiminnallisuutta ja suorituskykyä arvioitiin.

Tämän työn päätuloksena syntyi Smart-M3:n ja ASI:n välillä synkronointia varten virheensietokykyinen agenttisovellus, sekä sen arviointi. Sovellusta voi käyttää sellaisenaan, ja se tarjoaa myös pohjan uusien Aalto-palveluita hyödyntävien agenttien yhdistämiseen Smart-M3:een tai muuhun semanttiseen tietokantaan. Tämä helpottaa henkilökohtaisen Smart Space -käsitteen hyödyntämistä älykkääseen ja tiedon jakamiseen olemassaolevien internet-resurssien välillä, vaatimatta muutoksia niiden toteutukseen.

Avainsanat: Smart Spaces, RDF, RDF store, Smart-M3, ASI, REST, social networks

Acknowledgements

Writing this Master's Thesis has been an enlightening experience. I have learned a lot about writing and the technologies involved. All of this could not have happened without help from other people, however.

I want to especially thank my professor Heikki Saikkonen and instructor Mikko Vihonen for providing ideas, high level guidance, and asking the right questions; Olli Tyrkkö and Ian Oliver from Nokia Research Center for providing motivation and financial support for the project; Jukka Honkola and Hannu Laine for Smart-M3 technical support; Seppo Törmä and Esko Nuutila for helping me with ASI, synchronizing, Thesis writing, references, and proof reading; and Harri Hälikkä for providing continuous peer support and proof reading help throughout the project.

I would also like to thank Jari Heikkilä, Taina Hyppölä, Aleksi Hänninen, Juho Makkonen, Teemu Moisala, Vihtori Mäntylä, Henri Pihkala, Jukka Salminen, Timo Tuominen, Timo Töyry, Janne Vanhala, Mikko Viikari, many colleagues at Futurice Ltd., and the Python community for guidance, tips, fixes, and other forms of support during the project.

Special mentions go to my mom Leena Kontunen and current employer Futurice for flexibility, mental and material support, and Sampo Hämäläinen who arranged the project for this Thesis in the first place.

Helsinki, 3 June 2010

Eemeli Kantola

Contents

Abbreviations

API	Application Programming Interface
ASI	Aalto Social Interface
BOSH	Bidirectional-streams Over Synchronous HTTP
COS	Common Services (the old name for ASI)
CRUDS	Create, Read, Update, Delete, Subscribe
DIEM	Devices and Interoperability Ecosystem
FOAF	Friend of a Friend
GPS	Global Positioning System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
KP	Knowledge Processor
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
MIDE	Multidisciplinary Institute of Digitalisation and Energy
NTP	Network Time Protocol
pubsub	Publish/Subscribe
OWL	Web Ontology Language

QoS	Quality of Service
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
REST	Representational State Transfer
RFC	Request for Comments (a memorandum published by the Internet Engineering Task Force)
ROA	Resource-Oriented Architecture
RPC	Remote Procedure Call
SIB	Semantic Information Broker
SIOC	Semantically-Interlinked Online Communities
SOAP	Simple Object Access Protocol
SSAP	Smart Space Access Protocol
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDDI	Universal Description, Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
W3C	World Wide Web Consortium
WAP	Wireless Application Protocol
WBXML	WAP Binary XML
WebDAV	Web-based Distributed Authoring and Versioning
WQL	Wilbur Query Language (or WilburQL)
WSDL	Web Services Description Language
WWW	World Wide Web

XML	Extensible Markup Language
XHTML	Extensible Hypertext Markup Language
XMPP	Extensible Messaging and Presence Protocol (formerly known as Jabber)
YAML	YAML Ain't Markup Language

List of Figures

List of Tables

Chapter 1

Introduction

There is a large number of different internet services that the users may want to follow and update. Aggregating the relevant information can be done by mashing it up in a single database (Ankolekar *et al.* , 2010); on the other hand, overlapping information should be kept synchronized between services. Building and maintaining individual links between the services does not scale up well when increasing the number of services, and thus introducing a more centralized approach might be necessary.

Nokia Smart-M3 provides a suitable information sharing infrastructure, based on the Semantic Web technologies. Smart-M3 builds upon a semantic database that provides a publish/subscribe capable interface for updating information and getting notified of changes in the subscriber's areas of interest.

Battle & Benson (2008) have studied how the Web could be enhanced to enable querying essentially non-semantic information with Semantic Web tools. However, updating data back to the Web still needs work, let alone keeping information synchronized between multiple services.

This Thesis approaches the issue from a different angle by providing and analyzing a concrete piece of software for synchronizing data between a RESTful Web Service and Semantic Web, with the future possibility of supporting other Web Service endpoints for keeping data in separate real-world services synchronized. Aalto Social Interface[1] (ASI) will be used as a sample RESTful Web Service endpoint for synchronization. ASI provides existing user data for testing the integration with Smart-M3.

The results of this thesis can be utilized for improving the semantic interoperability

[1]http://cos.sizl.org (referred 5 Apr 2010)

of REST-based services. The key paradigms are non-intrusively enhancing existing Web Services to offer server push capabilities with publish/subscribe messaging, and Resource Description Framework (RDF) for a flexible and extensible information storage model. This brings the benefits of loose coupling and allows for dynamically introducing new services to the system, and enabling modifications to the existing configurations.

1.1 Synchronization Challenges

Retrieving and combining information poses a number of problems. First, a problem arises from fetching and combining data represented in different formats by the information sources (Oliver & Honkola, 2009) that can not be directly controlled. The terminologies, or ontologies, must be shared semantically between services to enable combining and sharing information intelligently.

Another question is how to detect changes in data in the external sources. Many of the existing internet services do not provide any way to automatically get a notification of the changes in data. When no inherent notification support is available, it is necessary to resort to polling, or client pull, for simulating subscription to changes. In any case, a method for loading only the relevant changes would be necessary for performance reasons.

Thirdly, data synchronization has to be managed in a proper way. Conflicts may arise in two-way synchronization of non-monotonically changing data. Sensible conflict resolution strategies need to be developed to manage different cases.

Finally, scalability needs to be taken into account. Even though the amount of data in localized Semantic Webs may not be an issue, support for monitoring changes in several different external information sources would be needed. The sources may not scale up for handling a large number of simultaneous update requests, and potentially an even larger number of subscriptions for information changes. In case of polling, the network and processor usage increases proportionally to the number of monitored pieces of information. If the updates occur frequently, the amount of data needed for synchronizing can also become an issue.

1.2 Goals for This Thesis

There are three primary goals in this Thesis:

1. designing and implementing a solution to synchronize data between Google OpenSocial-based ASI REST interface and Smart-M3's semantic database;
2. testing and using Smart-M3 and its Python knowledge processor interface library by writing a working application on top of it; and
3. implementing and evaluating an external, non-intrusive adapter that mediates between a traditional Web Service and a service using the publish/subscribe messaging paradigm.

1.3 Research Methods

The research was conducted by implementing a knowledge processor (KP) agent in the Python programming language for synchronizing data between the ASI interface and Smart-M3. ASI's hierarchical data structure had to be mapped onto a graph-like ontology, as required by the Resource Description Framework (RDF) representation in Smart-M3. The agent functionality was demonstrated with ASI's person structure and sample person data. Finally the application was analyzed qualitatively and quantitatively considering the original goals and perceived challenges.

1.4 Structural Summary

First this Thesis presents the existing research related to distributed systems and publish/subscribe, Web Services, and Semantic Web technologies including Smart Spaces. Then the implementation details are discussed at an architectural level, after which the results are analyzed. Lastly the conclusions and proposals for future studies are presented.

Chapter 2

Distributed Systems

2.1 Fundamentals

2.1.1 Definition

Ghosh (2006) characterizes distributed systems to consist of multiple, independent processes with disjoint address spaces, communicating via message passing and having a collective goal.

Distributed systems were originally developed to enable efficient resource sharing and thus decreasing costs. Later on the sharing of information across a large audience has increased greatly in importance, e.g. in the form of the internet, which itself is a huge distributed system. (Coulouris *et al.* , 2005, chap. 1)

2.1.2 Challenges

Distributing a system has the consequences of concurrent program execution, lack of global clock, and the components being susceptible of failing independently, without the others getting directly informed about failure states. The challenges regarding distributed systems are categorized as follows by Coulouris *et al.* :

- *Heterogeneity* of components. The different types of networks, hardware, operating systems, programming languages, and multiple developers across the system introduce complexity.
- *Openness* for extension through public, documented interfaces. Offering the

possibility for creating new components by different developers also adds complexity to the system design.

- *Security* concerns. Publishing the resources to be easily available introduces additional value to the users, but also potentially new ways for unauthorized parties to gain access to sensitive information.
- *Scalability* requirements. Being able to increase the system's resources and number of users significantly while still remaining effective requires additional measures.
- *Failure handling.* The problems lie first in detecting failures and then masking, tolerating or recovering from them to keep the system as highly available as possible.
- *Concurrency* in resource access. Allowing only a single client to access a resource at a particular time would severely limit throughput in many cases. Increasing concurrency requires additional synchronization techniques to maintain data consistency.
- *Transparency* or concealing the individual details behind a common interface. The user of a distributed system should not need to know the internal implementation details in order to use it. Examples include accessing the system interfaces without knowing the system's physical location, or transient error conditions.

The complex nature of the distributed systems suggest that a systematic approach be useful in tackling the problems involved (Coulouris *et al.* , 2005).

2.1.3 Design Requirements

When designing a distributed system, several non-functional requirements need to be taken into account. The main requirements that are relevant for most distributed systems, according to Coulouris *et al.* (2005), are described below.

Performance

The system's performance has to remain at sufficient levels also with high levels of concurrency and load. The delay between an external interaction and the consequent response from the system should be small enough and consistent. For instance, when loading a page with a web browser, long response times have a negative impact on the user experience.

Quality of Service

The perceived quality of service (QoS) is affected by several factors, of which the main ones are reliability, security and performance. Also adaptability for change can be a major factor if the system is to be deployed in a dynamic environment or faces several changing requirements.

Use of Caching and Replication

In many cases, a major factor helping satisfy the performance requirements is the introduction of caching temporary snapshots of a representation that has been built using dynamic data. Furthermore, replicating data is useful for reducing bottlenecks in the distribution channel.

Dependability

Dependability is comprised of program correctness, security aspects, and fault tolerance; i.e. how much the system can be relied on. Dependability is of high priority in business-critical systems.

2.2 Architecture Models

Several different style architecture models can be defined for distributed systems. An architecture model describes how the different parts of a system are placed in relation to each other, and how they interact (Coulouris *et al.* , 2005).

2.2.1 Architecture Layers

The distributed system architecture can be broken down into three main horizontal layers, in the increasing order of abstraction: platform, middleware, and application/service layers, as shown in Figure 2.1.

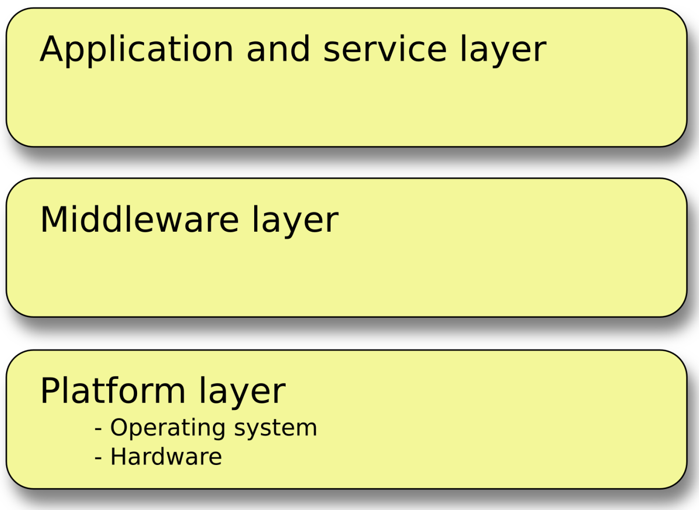


Figure 2.1: Distributed system's horizontal architecture layers

Platform Layer

Platform is the system's lowest level layer, consisting of the hardware computer and network hardware components and the operating systems running on top of them.

Middleware Layer

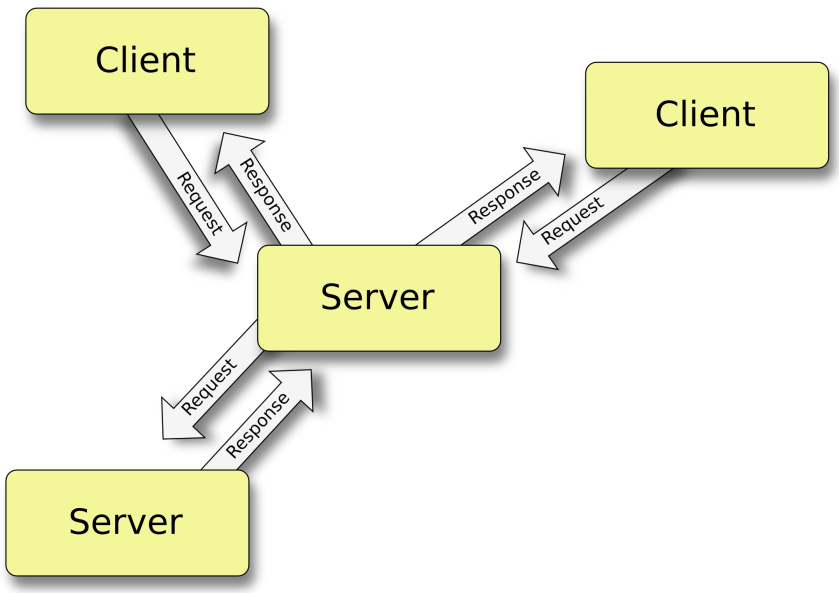
Middleware provides an abstraction of the platform layer, aiming at masking the heterogeneity of the underlying systems.

Application and Service Layer

This is the layer for software and services aimed at the end users, which may be either humans or other systems.

2.2.2 System Architectures

The two basic distributed architecture models according to Coulouris *et al.* (2005, chap. 2) are the *client-server* and *peer-to-peer* models. The client-server model consists of two different types of running programs, or processes:



[!h]

Figure 2.2: Sample client-server architecture

- *servers* that passively offer services by listening to service requests, then respond to them in trying to fulfill them in a specified way, and
- *clients* that actively initiate requests to the servers in order to send or get information, or a required operation to be performed; i.e. invokes an operation.

The same process can be a client and a server at the same time: in many cases, servers make use of other servers when fulfilling the requests. The terminology "client" and "server" is often clear in single request-reply pair's context where only the requesting and serving sides are involved. When referring to clients and servers in the system architecture's context, however, a clear distinction between client and server systems can usually be defined, even though many server processes also act as clients to other servers. Figure 2.2 depicts a sample client-server architecture.

Peer-to-peer systems are composed of processes that have no clear distinction between client or server processes, but the peer processes interact cooperatively and act as either clients or servers when needed.

Other useful system architecture models exist that are composed of the two basic models, such as *proxy*, *load balancer*, and *distributed hash table*.

2.3 Fundamental Models

The fundamental models for distributed system according to Coulouris *et al.* (2005), namely interaction, failure, and security, describe the common characteristics of components from which systems are constructed. This Section discusses the fundamental models in more detail.

2.3.1 Interaction Model

There are two main properties that have an effect on the process interaction in a distributed system: communication performance and the notion of time across different parts.

Communication performance is affected by delays or latency, channel bandwidth, and the variation of delay in delivering messages, i.e. jitter. These have to be taken into consideration when designing systems.

Clocks and timing is an issue because it is impossible to have the clocks of different components of a distributed system maintain the exactly same value over time, i.e. the clocks drift from perfect time. This problem can be mitigated, however, by using a common external time source such as a time server or Global Positioning System[1] (GPS) data to keep the clocks synchronized. Even this can only have the time synchronized to a certain degree.

The variations in timing have lead to different interaction models of distributed systems. Hadzilacos & Toueg (1994) define a synchronous distributed system to be one that:

- has lower and upper bounds for executing process steps,
- receives transmitted messages inside a known bounded time, and
- has a local clock with a drift rate that has a known bound regarding the real time.

An asynchronous system does not have any known bounds on these properties. Partially synchronous systems have bounds on some but not all of the properties. (Coulouris *et al.* , 2005, chap. 2 & 12)

[1]http://tycho.usno.navy.mil/gpsinfo.html (referred 29 Apr 2010)

2.3.2 Failure Model

Distributed systems introduce several different points of failures, compared to non-distributed systems. Failures can occur in processes, communication channels, or timing. Additionally arbitrary ("Byzantine") failures are also possible. Failure detection and management is of high importance in distributed systems in order to ensure correct functionality. (Coulouris *et al.* , 2005, chap. 2)

Process failures are failures that happen inside individual software processes across the distributed system. A process failure may cause the process to crash, i.e. completely stop functioning, or exhibit erroneous behavior such as outputting error messages.

Many transient error conditions can be detected and handled appropriately before letting the software crash. To prevent process failures in general from affecting the whole system's functionality, process monitoring can be set up to detect crashes or error conditions. The monitor can then trigger corrective measures like restarting the defunct process or escalating the failure condition to a higher level, for example causing a failure notification message to be sent to the party responsible for maintaining the process.

Communication Channel can fail in several different ways. The information sent may be corrupted during transfer, either not ever reach its destination or reach very slowly, or be duplicated. The reasons for communication failures can be faulty network hardware, network congestion and breakages.

Communication channel failures can be reduced by using a fault-tolerant data transmission protocol such as the Transmission Control Protocol (TCP; Postel, 1981) that introduces checksums to detect data corruption, timeouts and retransmission model to account for lost or corrupted data packets, and packet sequence number based accounting to detect duplicated or missing data packets. Also architecture models exist for supporting graceful communication failure handling, such as retransmitting messages in case of safe and idempotent operations in Resource-Oriented Architectures described later in Subection 3.3.1.

Timing failures mean wrong or non-uniform perceived ordering or timestamps of events affecting a distributed system or messages that are sent between the system's components. Timing can fail because of unpredictable delays in communication channels or clock skew, i.e. time difference of clocks between the system components. Using TCP on the transport level may introduce more delays since the protocol is optimized for accurate rather than timely delivery.

Timing failures can be reduced by or in some cases eliminated for all practical purposes by using a predictable communication channel and keeping the clocks synchronized (see 2.4.1 below). When this is impossible and more accuracy is needed, using algorithms such as the vector clock (Fidge, 1988; Mattern, 1989) can be utilized to determine message and event ordering to a degree, or detect situations where ordering is not possible, after which a selected conflict resolution strategy can be applied.

Arbitrary (Byzantine) failures are situations where components of the system fail by processing requests incorrectly, corrupting data, and/or providing erroneous or inconsistent output that looks otherwise valid as opposed to explicit error messages that result from detectable process failures. After a Byzantine failure, the system may respond in an arbitrary way. Byzantine failure causes include faulty hardware, human error, or malicious attacks.

Arbitrary failures are the most difficult to detect and correct. In many classic agreement problems, Byzantine fault tolerant (BFT) algorithms can ensure correct operation only if fewer than one third of the processes are faulty. Practically feasible algorithms exist, however, as first shown by Castro & Liskov (2002) and later implemented in the form of various protocols and frameworks, such as UpRight[2].

2.3.3 Security Model

Modular and open distributed systems expose themselves for external and internal threats. The security model describes the forms of the attacks, providing a foundation for analysing and designing system security.

Security can be analyzed by considering resource protection, processes and interactions (message passing), or evaluating different threats from the enemy, such as threats to the processes, communication channels, or the effectiveness of a denial-of-service attack.

[2]http://code.google.com/p/upright (referred 5 Apr 2010)

Communication channels can be secured by using cryptography and secret keys shared between the communicating processes. Authentication, i.e. methods for verifying if the other end is the one it is supposed to be, can be utilized whenever the information source needs to be trustworthy.

Denial-of-service attacks are more difficult to counter, as the messages to be used for such could be formally valid, albeit otherwise meaningless.

Threats can be modeled by composing a list about different forms of attacks, and then evaluating risks associated with them.

2.4 Interprocess Communication

Coordinating activities between processes requires communication between them. This poses a number of problems, such as:

- What is the whole system's state at a given time?
- Which process was the first to act?
- How do the processes communicate and understand each other?
- How are the communication channels constructed using standard web technologies?

The interaction and failure models presented in Section 2.3 provide several means for approaching the first two questions, which are also discussed in more detail in the following subsection 2.4.1. Furthermore, the last two problems are elaborated in the rest of this Section by introducing the marshalling and publish/subscribe concepts; the following Chapters 3 and 4 delve deeper into these questions in the context of Web Services and Semantic Web.

2.4.1 Time, States, and Coordination

The concept of time is problematic in distributed systems, because it is even theoretically impossible to keep clocks perfectly synchronized across the system. This has several important consequences related to timestamping and ordering of events. (Coulouris *et al.* , 2005, chap. 11)

First, it is not possible to assign event timestamps that would be agreeable to all parts of the distributed system. If an event that occurs in a system component is made known to another receiving component, the exact time on which the event affects the both components is necessarily different because of communication delays. On the other hand, even if the event would be appropriately timestamped in the original source component, this timestamp could in some cases denote a later time than the receiving component's perception of time because of *clock skew*, i.e. the difference in time shown by the clocks. The event would seem to have happened in the future, which contradicts causality.

Other difficulties arise from the perception of system state. If an event occurs that would modify the whole system's state, this information should be updated all over the system. This update will not be instantaneous, however. Additionally, problem arises when the system state is modified nearly simultaneously but in conflicting ways in different parts of the system. The conflict situation would need to be resolved. Otherwise the system will be left in an inconsistent state, which could have detrimental, far-reaching effects e.g. in case of financial applications.

Several measures have been devised to manage the coordination. Timing issues can be alleviated by using logical clocks (Lamport, 1978) for event ordering, and for time synchronization e.g. Network Time Protocol (NTP) (Mills, 1995), to maintain the clocks approximately synchronized. For managing system state, message-passing based election solutions exist for eventually achieving consensus (Coulouris *et al.* , 2005, chap. 12).

2.4.2 Marshalling

Marshalling is the process of converting data items in memory into a representation form suitable for storage or transmission as a message. The opposite process is called unmarshalling. (Coulouris *et al.* , 2005, chap. 4)

Marshalling is applied before sending the data in messages between distributed systems. There are several possible representation formats for marshalled data, such as the human-readable XML, JSON and YAML formats (see below for details), along with the binary Hessian[3], WAP Binary XML[4] (WBXML), and Java's serialized forms, among others. In this Thesis, XML and JSON are the two representations

[3]http://hessian.caucho.com (referred 1 Jun 2010)
[4]http://www.w3.org/TR/wbxml (referred 1 Jun 2010)

used, and are described below in more detail.

Marshalling is considered synonymous to serialization by Python's *marshal* module[5]. However, RFC 2713[6] gives the distinction that marshalling records the object's source code, whereas serialization does not. For the purposes of this Thesis, Python's definition will be used, i.e. only the object's state and not code is considered.

XML

XML (Extensible Markup Language) defines a textual format for structured data. The World Wide Web Consortium (W3C) defines XML as follows (Bray *et al.* , 2008):

> XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

XML's design focuses on documents, but it is widely used in representing arbitrary data structures, for example in Web Services (see Chapter 3).

JSON

JSON (JavaScript Object Notation) is a lightweight, text-based, human-readable data structure format, originally specified by Crockford (2006) in RFC 4627. Despite its name and syntax that reflect the ECMAScript Programming Language Standard, JSON is language-independent.

JSON can represent the primitives string, number, boolean, and null, and additionally the structured types object and array. Being specifically a data interchange format, JSON is syntactically simpler and smaller in size compared to the general-purpose XML.

[5]http://www.python.org/doc/lib/module-marshal.html (referred 14 Mar 2010)

[6]Schema for Representing Java(tm) Objects in an LDAP Directory, http://tools.ietf.org/html/rfc2713 (referred 14 Mar 2010)

The simplistic JSON lacks support for relations inside a document, extensibility, and the easier human readable block syntax format that JSON's superset YAML[7] provides.

2.4.3 Publish/Subscribe

Publish/subscribe (pubsub) is an asynchronous messaging paradigm that enables loosely coupled interaction in dynamic applications. Subscribers express their interest to specific events, which the publishers generate and notify about. The subscriptions have different variants, such as topic-based, content-based, or type-based. The event-based interaction fully decouples the systems in time, space, and synchronization. (Eugster *et al.* , 2003)

Pubsub model relies on *server push* technology where the server is capable of initiating communication with clients. Methods for introducing push capabilities on the Web include:

- *HTTP server push* (or HTTP streaming) where connection is kept open for more updates after the normal request-response sequence. This approach is problematic when serving web browsers, since not all of them support HTTP server push in the same way. However, enabling technologies like *Server-Sent Events*[8] or *Web Sockets*[9] are being standardized as a part of HTML5.
- *Long polling* which emulates server push with standard polling. The server keeps the response to the original request pending and sends response only after new updates are available, after which the client sends a new request immediately. For instance, the draft standard *BOSH*[10] from XMPP Standards Foundation[11] employs long polling for providing server push support.

Pubsub-Mediators

Mediator design pattern (Gamma *et al.* , 1994) is about encapsulating communication between objects with an additional mediator object. It aims in reducing

[7]YAML Ain't Markup Language (YAML™) Version 1.2, http://yaml.org/spec/1.2/spec.html (referred 14 Mar 2010)
[8]http://dev.w3.org/html5/eventsource (referred 27 Apr 2010)
[9]http://dev.w3.org/html5/websockets (referred 29 Apr 2010)
[10]XEP-0124: Bidirectional-streams Over Synchronous HTTP, http://xmpp.org/extensions/xep-0124.html (referred 29 Apr 2010)
[11]http://xmpp.org (referred 29 Apr 2010)

dependencies between the communicating parties, thus lowering coupling across the system.

Pubsub-mediators are mediators between a pubsub-capable interface and other services. Pubsub middleware components as an integration point facilitate building composite services. The mediators interact with the non-pubsub-capable component services and communicate with the pubsub middleware.

The advantage of the pubsub mediators is that they can add pubsub capabilities to a non-pubsub-aware service in a dynamic manner, without requiring changes to that service's implementation. Monitoring updates in this kind of services might require *client pull*, or polling, where the client actively checks for data updates on the service at regular intervals.

Chapter 3

Web Services

3.1 Definitions

The World Wide Web (WWW), commonly known as the Web, is a system of interlinked hypertext documents on the Internet. WWW has been designed with flexibility and a non-centralized architecture in mind, and the chosen base technologies, to be described below, support these properties. (Berners-Lee, 1996)

Web Services commonly refer to clients and servers that communicate over the Hypertext Transfer Protocol (HTTP) used on the Web. Contrast this with web services (without capitalization), which could refer to, for instance, any non-internet service advertised via WWW. The term Web Services is, however, ambiguous with several different uses and definitions.

The World Wide Web Consortium (W3C) provides the following definition[1]:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Alonso *et al.* (2004) refer to the following Universal Description, Discovery and Integration (UDDI) consortium's definition.

[1]Web Services Glossary, http://www.w3.org/TR/ws-gloss (referred 21 Feb 2010)

> Web services are self-contained, modular business applications that have open, internet-oriented, standards-based interfaces.

Richardson & Ruby (2007) define the Web Services to be simply applications on the WWW, i.e. clients and servers that communicate over the HTTP protocol. This definition is henceforth considered when referring to the term Web Services throughout this Thesis.

3.1.1 Web Service Categorization

Richardson & Ruby divide Web Services into two categories:

- *Big Web Services* use the traditional WS-* stack, referring to e.g. WSDL, SOAP, and WS-Security standards[2] and often resorting to remote procedure calls (RPC). Big Web Services are versatile and provide powerful abstractions, but bring a large amount of additional complexity to service adaptation, scaling, and maintenance.
- *RESTful Web Services* emphasize simplicity and build on the existing, well-known Web technologies URI and HTTP. The benefit of utilizing RESTful web services is existing software and internet infrastructure support. Furthermore, RESTful services are more easily scalable because of avoiding the need to maintain and, in case of distributed Web Services, synchronize the user session information in the servers.

In this Thesis, the Big Web Services are not considered in more detail, but are mentioned here for completeness. The following Sections concentrate on elaborating the RESTful Web Service technologies.

3.2 Web Technologies

The Web technologies relevant for Web Services are HTTP as the communication protocol between servers and clients, URLs with the “http” scheme to locate internet resources, and common representation formats, which the following Subsections discuss in more detail.

[2]See Web Services Architecture, http://www.w3.org/TR/ws-arch (referred 22 Apr 2010), for an overview

3.2.1 Uniform Resource Identifier

A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource.

Uniformity means that URIs can denote different kinds of resources with a consistent semantic interpretation, enables extending the URI space without interfering with the existing ones, and permits new applications or protocols to build on the existing set of URIs. Resource can be an abstract or concrete concept, and the URI specification does not limit the scope of what a resources should be. An identifier distinguishes a resource that is being identified from other identifiable resources. (Berners-Lee *et al.* , 2005)

The generic URI syntax defines URI character sequence to consist of a scheme designator string and a scheme-specific identifier which are separated by a semicolon: *scheme : scheme-specific-identifier.* A URI can refer to an absolute or a relative identity.

A URI can be further classified as a locator (URL), which is a common concept in WWW, or other subclasses such as a name (URN), used for identification.

Uniform Resource Locator is a compact string representation for a resource available via the Internet (Berners-Lee *et al.* , 1994). URLs are a subset of the more general URI specification. For Web Services, “http” and “https” are the most commonly used schemes for accessing resources.

3.2.2 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, extensible, and stateless protocol, and has also many uses beyond hypertext. HTTP provides support for negotiating data representation formats. The current HTTP version in wide use is 1.1. (Fielding *et al.* , 1999)

HTTP communication is based on requests sent by clients, and responses sent back by servers. The main component of a HTTP request is the request line with method and resource descriptions, which is followed by request headers and an optional request body. The resources to be accessed are identified by URLs with the “http” scheme. An HTTP response contains one of the HTTP response codes, followed by response headers and possibly a body.

HTTP protocol does not inherently provide encryption, which is however attainable by using the *Hypertext Transfer Protocol Secure* (HTTPS; Rescorla, 2000). HTTPS utilizes the Transport Layer Security (TLS) protocol and its predecessor, Secure Socket Layer (SSL), to encrypt the communication channel. HTTPS URIs are denoted by the “https” scheme.

3.2.3 Representation formats

The information stored in the Web needs to be accessible for the consumers, be it humans or computer programs. The data representation format plays an important role in making the information contained available. Human-readable Web pages are predominantly formatted in one of the HyperText Markup Language (HTML) variants[3]. On the other hand, Web Services typically utilize marshalling information to be able to communicate with more rigidly structured formats such as XML and JSON, described earlier in Subsection 2.4.2.

3.3 Representational State Transfer

Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. The term REST was coined by Fielding (2000) in his doctoral dissertation.

REST is the lightweight alternative to the Big Web Services, largely relying on existing web technologies, HTTP and URI. REST defines six constraints: client-server model, statelessness, cacheability support, uniform interface, system layering, and the optional code-on-demand constraint (Fielding, 2000, chap. 5). A service conforming to the REST constraints is referred to as being *RESTful*.

3.3.1 Resource-Oriented Architectures

REST is not an architecture per se, but a set of design criteria that leaves a lot of room for interpretation; for instance, a REST-RPC hybrid architecture could

[3]Most commonly HTML 4.01, XHTML, or the upcoming HTML 5 – see http://www.w3.org/TR/html401, http://www.w3.org/TR/xhtml1, and http://www.w3.org/html/wg/html5/ (referred 22 Apr 2010)

conform to REST's requirements. Resource-Oriented Architectures (ROA) clarifies the concept by defining the architecture and providing additional constraints for it. (Richardson & Ruby, 2007, chap. 4)

The main elements ROA involves in are resources, names, representations, and links, on which the architecture builds. The architecture also defines the following properties that need to hold in order to conform to ROA.

- Addressability. The resources need to be identifiable by distinct addresses, such as URIs.
- Statelessness. The application state remains only on client and may not be stored on the server. A request contain all the information that is needed for carrying it out. HTTP requests happen completely isolated from each other. Note that the statelessness concept refers to only not storing client's state on server side. The server and client can always have a state (resource state and application state), but the distinction is that the client knows its whole state and sends it to the server on every request.
- Representation. The representation format, if selectable, is to be requested e.g. in the URIs (by appending "?lang=en") or in HTTP "Accept" headers. The server then decides the appropriate representation format, taking the client preferences into account.
- Connectedness. The resources have to be linkable, e.g. using URIs. A simple Web analogue of the connectedness property would be a search engine result page, with links containing all the information required for retrieving the search results.

The Uniform Interface

Moreover, HTTP protocol leaves some room for interpretation regarding the different HTTP methods, which ROA aims to reduce. Especially HTTP POST has previously been problematic because it's definition doesn't give clear constraints for usage, unlike the other methods. The usage of HTTP methods defined by ROA is the following (Richardson & Ruby, 2007):

- Retrieve a representation without modifying resources: GET;

- Create a resource: PUT in case of a new but known URI, POST if creating a resource without knowing the new URI beforehand – this separation is widely misunderstood;
- Modify a resource: PUT to an existing URI;
- Delete an existing resource: DELETE.

Additionally, the HEAD (retrieve metadata) and OPTIONS (check what operations are allowed) methods are allowed by ROA.

Other methods, such as the MOVE, COPY, and SEARCH of the WebDAV[4] extension, are allowed by HTTP but not considered part of ROA as defined by Richardson & Ruby.

Safety and Idempotence

Because of ROA's additional definitions, two useful properties can be defined for the ROA operations. An operation is:

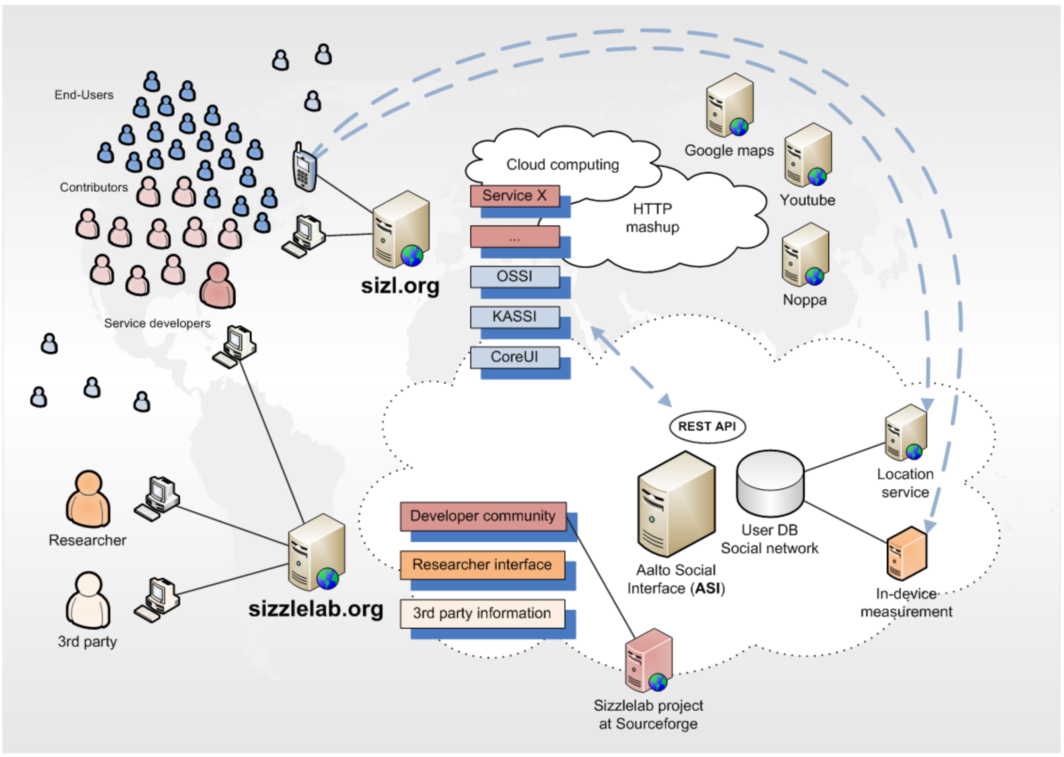
- *safe* if it does not cause modifications to any resources; and
- *idempotent* if no subsequent invocation after the first one cause modifications to the resources.

Therefore it can be maintained that GET, HEAD and OPTIONS are safe, and PUT and DELETE in addition to the safe operations are idempotent in ROA. POST is neither safe nor idempotent. This observation brings additional freedom with ROA applications when dealing with safe and idempotent operations, since they can be repeated without endangering data integrity even when connection failures are possible.

3.4 Aalto Social Interface

Aalto Social Interface[5] (ASI) is a RESTful platform for social media applications. ASI is a part of the OtaSizzle research project[6] whose goal is to develop an open

[4]http://www.ietf.org/rfc/rfc4918.txt (referred 23 Apr 2010)
[5]http://cos.sizl.org (referred 5 Apr 2010)
[6]http://mide.tkk.fi/en/OtaSizzle (referred 5 Apr 2010))



© 2010 OtaSizzle (MIDE/TKK); source:
http://www.sizzlelab.org/content/otasizzle-overview (referred 11 Apr 2010)

Figure 3.1: ASI in OtaSizzle's high level architecture

experimentation environment for testing mobile social media services. ASI has been previously known as Common Services (COS). Figure 3.1 shows OtaSizzle's high level architecture and how ASI positions itself in it.

ASI provides some but not all of the features specified in OpenSocial, a set of APIs for distributed social applications Google (2009). There are several differences, though: for instance, the `updated` field for a person object is called `updated_at` in ASI. ASI also uses HTTP cookie based sessions for authenticating clients, whereas OpenSocial requires OAuth[7]. Nevertheless, ASI can be regarded as a ROA application except perhaps regarding the session handling.

Internally, ASI has been built using the *Ruby on Rails*[8] web application framework.

There are several applications built on top of ASI, such as Kassi[9], a social market-

[7]http://oauth.net (referred 5 Apr 2010))
[8]http://rubyonrails.org (referred 30 Apr 2010)
[9]http://kassi.sizl.org?locale=en (referred 5 Apr 2010)

place for items and favors.

Chapter 4

Semantic Web

4.1 Concepts

Semantic Web is a web of information with meaning, allowing for the interoperability of the systems by logically connecting different terms to each other. The Semantic Web is also a non-centralized knowledge representation system, utilizing XML (described earlier in Subsection 2.4.2) for structuring information, and Resource Description Framework (RDF) for expressing meaning. (Berners-Lee *et al.* , 2001)

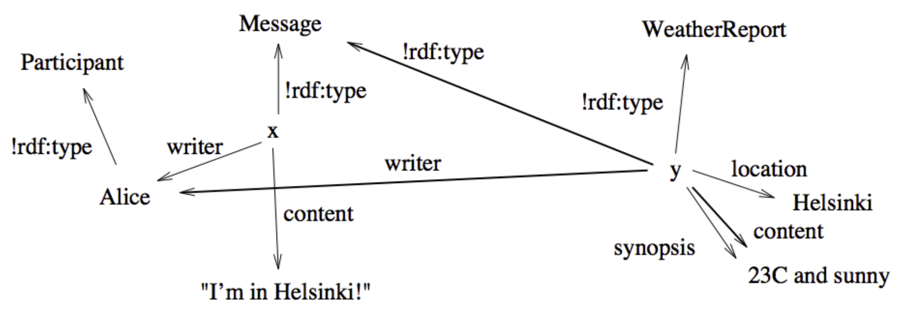
As noted by Shadbolt *et al.* (2006), the Semantic Web had not experienced large-scale deployment. There exist, however, several specific encouraging applications such as applying Semantic Web technologies to government and aviation sector data (Alani *et al.* , 2008). On the other hand, enhancing the Web with semantic features, e.g. by taking advantage of HTML5 Microdata draft specification[1], could prove useful if adopted by the Web community.

4.1.1 Resource Description Framework

The Resource Description Framework (RDF) is a collection of World Wide Web Consortium[2] (W3C) specifications. RDF defines a general-purpose language for modeling concepts and representing information in the Web.

[1]http://whatwg.org/specs/web-apps/current-work/multipage/links.html#microdata (referred 24 May 2010)

[2]http://www.w3.org (referred 21 Apr 2010)



© 2008 Ian Oliver and Jukka Honkola; source: Oliver & Honkola (2008)

Figure 4.1: RDF sample graph

RDF encodes meaning in sets of triples, containing a subject, a predicate (verb), and an object. This can be used as a natural way of describing most of the machine-processable information. The possible types for subject are URI (see 3.2.1) and *blank node* or "anonymous resource". Predicate can be a URI or blank node, and object can additionally contain a *literal* value.

An RDF model represents a labeled, directed multi-graph that enables evolutionally developable datasets. Being able to extend the data model becomes necessary, when it is difficult or impossible to define it in advance at a sufficient level.

Figure 4.1 shows a sample RDF graph.

RDF can be represented in several ways, such as RDF/XML (Beckett & McBride, 2004), the better human-readable Notation3[3], or RDFa[4] ("RDF in XHTML: Syntax and Processing", W3C working draft) for HTML5 integration.

The storage for RDF triples is called a *triplestore*.

4.1.2 Ontologies

An ontology is a description of what types of things exist. This can include a taxonomy description and inference rules between different entities. Taxonomy is used for defining hierarchical data structures, composed of object classes and their relations. Furthermore, inference can be used to deduce relations not directly expressed by the taxonomy, such as if object A is of type B and B is of type C, then A must also be of type C.

[3]http://www.w3.org/DesignIssues/Notation3.html (referred 21 Apr 2010)
[4]http://www.w3.org/TR/rdfa-in-html (referred 24 May 2010)

Several ontologies have been standardized, such as the social networking related Friend of a Friend[5] (FOAF), part of W3C's Semantic Web project[6], and Semantically-Interlinked Online Communities[7] [8] (SIOC).

RDF Schema

RDF Schema is an extensible knowledge representation language, with elements for describing ontologies in RDF. The current RDF Schema specification[9] has been released by W3C in 2004.

RDF Schema defines several URIs to be used in describing ontologies, such as:

- Classes (rdfs:Class) and subclasses (rdfs:subClassOf)
- Properties (rdfs:property, rdfs:domain, rdfs:range, rdfs:subPropertyOf)
- Utility properties (rdfs:seeAlso, rdfs:isDefinedBy)

A simple example of RDF Schema usage would be to define a class, and then an object to be of the class (in Notation3 syntax):

```
@prefix : <http://some.uri#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:SomeType rdf:type rdfs:Class .
:thing rdf:type :SomeType .
```

This indicates that `http://some.uri#SomeType` defines a class of objects, and `http://some.uri#thing` belongs to that class.

Web Ontology Language

Web Ontology Language (OWL) is a knowledge representation language for publishing and sharing ontologies on the World Wide Web. It is a vocabulary extension

[5]http://www.foaf-project.org (referred 16 May 2010)
[6]http://www.w3.org/2001/sw (referred 16 May 2010)
[7]http://sioc-project.org (referred 16 May 2010)
[8]http://rdfs.org/sioc/spec (referred 16 May 2010)
[9]http://www.w3.org/TR/rdf-schema (referred 21 Apr 2010)

to RDF. The current version is OWL 2[10] that is backwards compatible with earlier specifications.

4.1.3 Agents

In the context of Semantic Web, agents, also known as knowledge processors, are pieces of programs that consume and produce semantic information. Agents provide the means for actually taking advantage of the information contained in Semantic Web.

Examples of agents include reasoners that produce more data based on existing semantic information, end user applications built on top of Semantic Web, or mediators that supply and adapt information to and from other sources.

4.2 Smart Spaces

Global ontology for the Semantic Web seems difficult if not impossible to create, and an anarchistic model could be more realistic. Oliver (2009) maintains that the future of Semantic Web lies in local, personal and individually evolving information spaces, and not so much in its global forms earlier envisioned by Berners-Lee *et al.* (2001). The Semantic Web is most succesful in specific, small-scale, domain-restricted situations, and this would be the main future direction of development.

The more localized nature of Semantic Web has given rise to concepts such as Smart Spaces, studied e.g. at Nokia Research Center[11]. Smart Spaces is an umbrella concept involving in context-aware communication and interoperability between devices and people. Some of the recent research include device interoperability (Lappeteläinen *et al.* , 2008), space-based computing (Oliver & Boldyrev, 2009) and their applications (Honkola *et al.* , 2009).

4.2.1 Nokia Smart-M3

Smart-M3[12] is an open source software project for Semantic Web based, distributed information sharing infrastructure. Smart-M3 provides a pubsub-capable interface

[10]http://www.w3.org/TR/owl2-overview (referred 21 Apr 2010)

[11]http://research.nokia.com/research/labs/nrc_smart_spaces_laboratory (referred 22 Apr 2010)

[12]http://smart-m3.sourceforge.net (referred 22 Apr 2010)

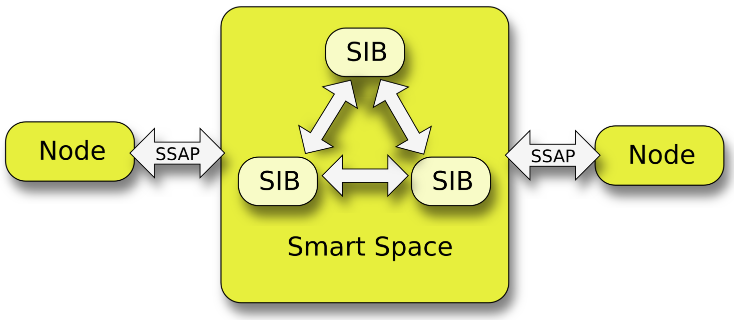


Figure 4.2: Smart-M3 architecture overview

for subscribing to and manipulating information stored as RDF triples. Smart-M3 has formerly been also known as Sedvice, M3, or Nokia M3.

Smart-M3's Smart Space consists of one or more *Semantic Information Brokers* (SIB) that communicate with external agent nodes using *Smart Space Access Protocol* (SSAP). See Figure 4.2 for a Smart-M3 architecture overview.

An example of Smart-M3 usage in a business meeting environment is presented by Oliver *et al.* (2009).

4.2.2 Semantic Information Broker

Semantic Information Broker (SIB) is one kind of a triplestore for storing, sharing and governing the information stored within it as RDF triples. For communicating with the data users, SIB provides an interface supporting Smart Space Access Protocol (SSAP), an XML-based communication protocol enabling the use of pubsub messaging paradigm.

SIB supports CRUDS (Create, Read, Update, Delete, Subscribe) operations, called *insert*, *query*, *update*, *delete*, and *subscribe* by Smart-M3. The creating, updating and deleting functionality relies on RDF triples as the data units, whereas queries and subscriptions can also utilize the more expressive *Wilbur Query Language* (WilburQL or WQL), discussed e.g. by Lassila (2007).

Smart-M3 with agents could be regarded as a kind of middleware (see 2.2.1) that masks the heterogeneity of the connected data components by aggregating them into a single semantic database.

Chapter 5

Implementation

5.1 Overview

The goals of this Thesis, namely creating a synchronizing mediator agent between ASI and Smart-M3 and then evaluating it from different aspects, were approached by first implementing the software. The implementation consists of a platform and a middleware layer, as introduced in Subsection 2.2.1. There are no components in the application layer, since the implementation is mainly focused on unifying the underlying data structures, to be then used by the actual end user applications that are to be provided separately. The actual applications and services would be built on top of ASI, and possibly SIB.

The following Sections in this Chapter describe and justify the software's architecture choices and then implementation concerns in more detail. First the platform choices are elaborated, and then the actual synchronization agent's external and internal architecture and detailed logic are discussed.

Notational conventions

In this Chapter, the following conventions are adhered to:

- *Emphasized* text is always used for program code library names, such as *asib-sync*; also when introducing new concepts and terms for the first time.
- `Typewriter` font is used for commands, files in filesystem, class, function or

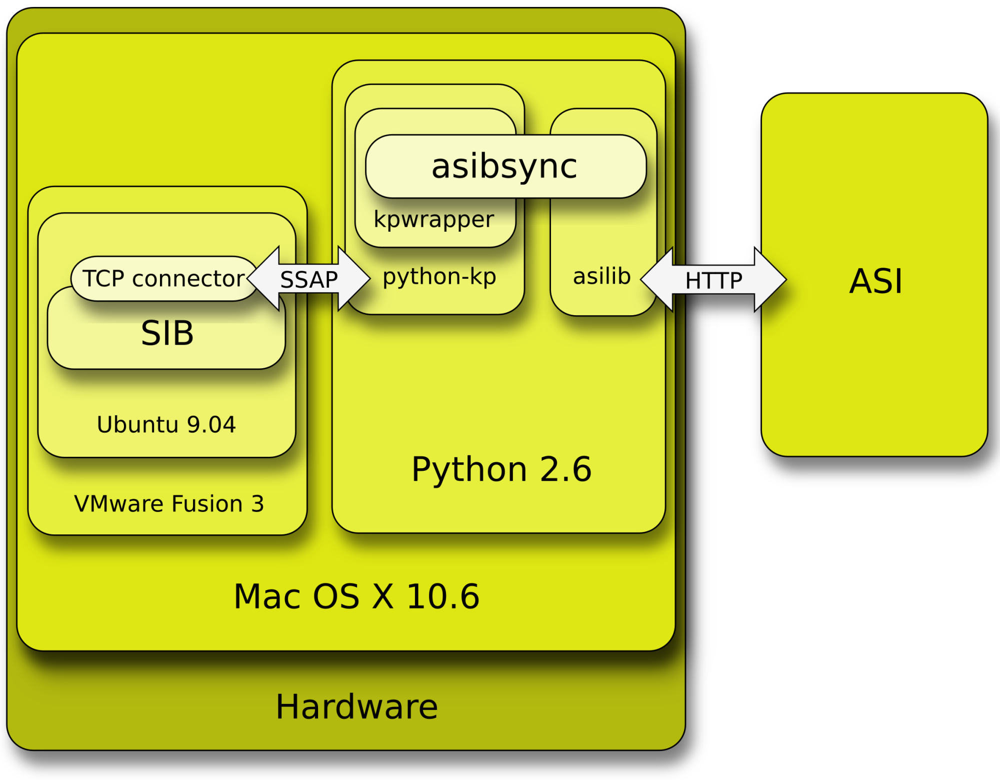


Figure 5.1: Overview of the platform and middleware architecture for the test setup

variable names in program code, and Python modules or packages; examples of the different uses include `sib-tcp`, `setup.py`, `ASIAgent`, and `asi_agent`, respectively.

5.2 Platform Architecture

Figure 5.1 gives an architectural overview of the platform and middleware level setup, which is described below in more detail.

The SIB knowledge processor library exists both as a C and a Python implementation. Of these two choices, Python was chosen because it was deemed more productive, as suggested by Prechelt (2000), and because using C for performance gains was not deemed necessary in this context. Python was also used for all other code for consistency and to avoid unnecessary extra interfaces between different modules in several programming languages.

Smart-M3 release version 0.9.2 was used for the implementation.

The Smart-M3 SIB daemon (`sibd`) and the accompanying SSAP over TCP interface (`sib-tcp`), both written in C, were compiled and run on Ubuntu 9.04 i386 desktop[1] installation, since Ubuntu is one of the operation systems Smart-M3 is advertised to be tested on according to the setup documentation[2]. SIB debug messages were turned on when compiling. The Ubuntu installation was running inside a VMware Fusion 3.0.2 virtual machine environment[3].

Rest of the used program code, solely consisting of Python libraries and applications, were run on Mac OS X 10.6.3 and Python 2.6.1 shipped with the operating system. The connection to SIB running under Ubuntu was made via a virtual local area network (LAN) interface provided by VMware.

Eclipse[4] IDE (Integrated Development Environment) with Aptana[5] PyDev[6] plugin were chosen as the development environment.

5.2.1 Python Terminology and Tools

In this Thesis, Python is the programming language of choice. Therefore a short introduction is in order to the basic Python terminology, necessary tools, and web resources to better understand the subsequent architectural choices and descriptions.

A piece of Python code, or *script*, is normally stored inside `.py` files in the filesystem. A single Python source file is called a *module*, and can contain any number of variables, functions and classes. Modules can be fully or partially imported in other modules and scripts, i.e. loaded from outside them, to take advantage of their functionality and resources.

Individual modules can be further structured into Python packages. They are fundamentally filesystem directories that contain at least a module called `__init__.py`, which can be an empty file. Packages are a convenient way for organizing modules into hierarchies and *namespaces*. Python keeps track of a system path for locat-

[1]http://releases.ubuntu.com/9.04 (referred 5 Apr 2010)
[2]http://sourceforge.net/projects/smart-m3/files (referred 5 Apr 2010)
[3]http://www.vmware.com/products/fusion (referred 5 Apr 2010)
[4]http://www.eclipse.com (referred 20 Apr 2010)
[5]http://www.aptana.com (referred 20 Apr 2010)
[6]http://pydev.org (referred 20 Apr 2010)

ing modules and packages. This path can be influenced e.g. via the `PYTHONPATH` environment variable.

See e.g. the Python tutorial's Modules chapter[7] for more insight into the modules and packages.

Python modules and packages can be distributed in *egg* format[8] that is a compressed archive accompanied with metadata describing its dependencies and code within. Python eggs can be managed by Python's standard *Distutils* library[9] and installed by invoking the `setup.py` contained. This process can be further automated. The different alternatives include `easy_install` from the *setuptools*[10] package, which is considered deprecated by some[11], or the replacements *Distribute*[12] or *pip*[13]. In more complicated setups where managing multiple eggs and version dependencies in isolation regarding other applications is necessary, build management and automation tools such as *Buildout*[14] can be utilized.

The unified way for distributing Python eggs to the community is to use Python Package Index, or *PyPI*[15]. The Distutils system provides a way to easily publish new and updated egg versions to PyPI through the `setup.py` script.

5.3 Synchronization Agent

The synchronization agent consists of three main components: the ASI client library (*asilib*), SIB knowledge processor library (*python-kp*) with a wrapper (*kpwrapper*) on top of it, and the actual ASI-SIB synchronization agent module (*asibsync*) that uses the client library stacks for interacting with ASI and SIB, respectively.

Additionally, two small command line helper libraries, *kpconsole* and *asiconsole*, were written for interactive testing and managing low level data in both ASI and SIB. Both utilize the *IPython*[16] library for providing a developer-friendly interactive console.

[7]http://docs.python.org/tutorial/modules.html (referred 5 Apr 2010)
[8]http://peak.telecommunity.com/DevCenter/PythonEggs (referred 5 Apr 2010)
[9]http://docs.python.org/library/distutils.html (referred 5 Apr 2010)
[10]http://pypi.python.org/pypi/setuptools (referred 5 Apr 2010)
[11]http://www.b-list.org/weblog/2008/dec/14/packaging (referred 5 Apr 2010)
[12]http://pypi.python.org/pypi/distribute (referred 5 Apr 2010)
[13]http://pypi.python.org/pypi/pip (referred 5 Apr 2010)
[14]http://pypi.python.org/pypi/zc.buildout (referred 5 Apr 2010)
[15]http://pypi.python.org/pypi (referred 5 Apr 2010)
[16]http://ipython.scipy.org (referred 5 Apr 2010)

All the three main components and two helpers are packaged as Python eggs. The code written for this Thesis has been open-sourced under a BSD-like license[17] and made available through a SourceForge repository, http://asibsync.sourceforge.net, and individual libraries in Python Package Index. A single buildout consisting of all the components is provided in the SourceForge code repository. Unit tests have been written for the most relevant algorithms.

More detailed description of the components follows.

5.3.1 ASI Library, *asilib*

The ASI interface library, *asilib*, was designed to be simple and easy to use also for interactive debugging, to facilitate developing the actual sync agent. *asilib*'s external interface consists of one class, `ASIConnection`, that has methods for generating ASI REST requests and returning the results as Python lists, dictionaries, or a combination thereof. See Program 1 for a short sample code that uses *asilib*.

asilib is a relatively thin wrapper on top of Python's existing *urllib2*[18], a collection of helper functions and classes for opening URLs (mainly HTTP). As *urllib2* can not handle all the HTTP methods as required by the ASI's REST interface, such as PUT and DELETE, support for these was added. Asilib also uses Python's *cookielib*[19] for managing ASI's HTTP sessions cookies. This is one of the aspects that differentiate the ASI interface from Google OpenSocial specification (see 3.4).

httplib2[20] was also considered instead of *urllib2* because of its better support for various HTTP methods, but was dropped because of the lacking cookie support.

5.3.2 SIB Knowledge Processor Wrapper Library, *kpwrapper*

The Smart-M3's Python knowledge processor library, or *python-kp*, is a relatively low level SIB interface library. It provides CRUDS transaction support and marshalling/unmarshalling tuple-based RDF representations to and from SSAP. However, *python-kp* lacks necessary abstraction that facilitate programming applications. To enable readable, easier-to-write code and interactive testing reasonably, a wrapper

[17]For a BSD license template, see http://www.opensource.org/licenses/bsd-license.php (referred 29 Apr 2010)
[18]http://docs.python.org/library/urllib2.html (referred 5 Apr 2010)
[19]http://docs.python.org/library/cookielib.html (referred 5 Apr 2010)
[20]http://code.google.com/p/httplib2 (referred 5 Apr 2010)

Program 1 Sample program using the *asilib* (included in *asilib* sources, `examples/asilib_sample.py`)

```
from asilib import ASIConnection

conf = {
    'base_url': 'https://cos.alpha.sizl.org',
    'app_name': 'myapp',
    'app_password': 'secret',
    'username': 'joeuser',
    'password': 'hidden',
}

uid = '12345678'
with ASIConnection(**conf) as uc:
    user = uc.get_user(uid)
    friends = uc.get_friends(uid)
    print('User %s has %i friends' % (user.username, len(friends)))
```

library, *kpwrapper*, was written on top of *python-kp*.

kpwrapper adds support for reading the SIB connection configuration from a file, better RDF Triple abstraction for *python-kp*'s Python tuple representation, and abstracts the connection with a SIBConnection class that adds convenience methods to implicitly open transactions for the SIB's basic CRUDS operations (see 4.2.1).

kpwrapper's main package, `kpwrapper`, has the init module, `kpwrapper/__init__`, whose main components are the `SIBConnection` manager interface class (elaborated below) and `Triple` for abstracting the RDF triple handling and conversions. In addition, there are helper classes such as `uri`, `literal`, and `bnode` for handling RDF type information, `TransactionWrapper` for managing *python-kp*'s SIB transactions, and various internal helper methods.

Program 2 provides a simple demonstration of how the interface could be used in applications.

SIBConnection

The `SIBConnection` class acts as the main interface for SIB, which can be seen in action by looking at the examples in Program 2. `SIBConnection` wraps the `python-kp`'s transactions and tuples inside more manageable objects.

Program 2 Sample program using the *kpwrapper* library (included in *kpwrapper* sources, `examples/kpwrapper_sample.py`)

```
from kpwrapper import SIBConnection, Triple

if __name__ == '__main__':
    with SIBConnection('SIB console', 'preconfigured') as sc:
        results = sc.query(Triple('Space', 'has', None))
        contents = [triple.object for triple in results]
        print('Space has %s' % contents)
        # prints "Space has []"

        sc.insert(Triple('Space', 'has', 'space_junk'))

        results = sc.query(Triple('Space', 'has', None))
        new_contents = [triple.object for triple in results]
        print('Now space has %s' % new_contents)
        # prints "Now space has [literal('space_junk')]"

        sc.remove([Triple('Space', 'has', x) for x in new_contents])
        results = sc.query(Triple('Space', 'has', None))
        contents = [triple.object for triple in results]
        print('Extra junk removed, now Space has again %s' % contents)
        # prints "Extra junk removed, now Space has again []"
```

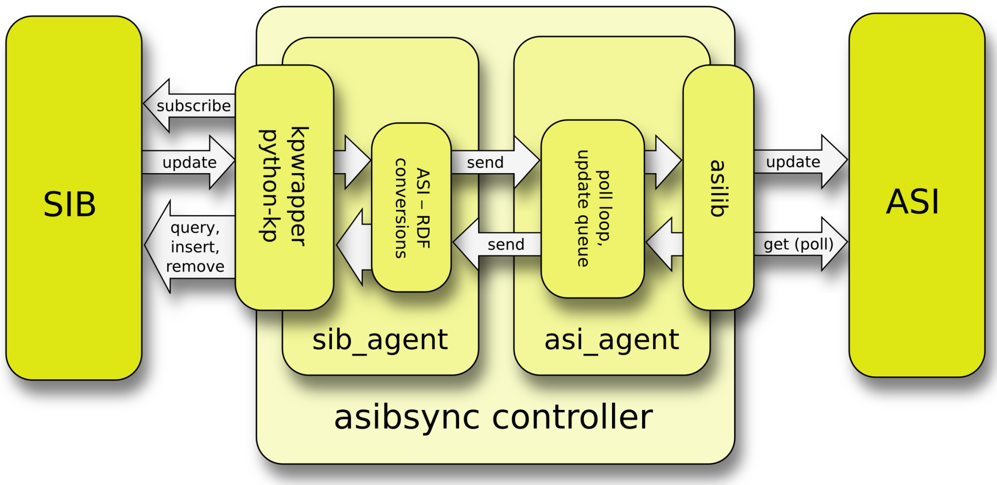


Figure 5.2: *asibsync* internal architecture and data flow diagram

5.3.3 ASI-SIB Synchronization Agent, *asibsync*

For synchronizing data between ASI and SIB, an agent application was written that utilizes *asilib* and *kpwrapper*. The agent, called *asibsync*, consists of three modules: `asi_agent` containing ASI polling functionality, `sib_agent` providing SIB connectivity, and the main `asibsync/__init__` controller module that starts up and connects both agents when run. The more detailed *asibsync* architecture is depicted in Figure 5.2.

The current implementation supports synchronizing one ASI's `person` object with a given identifier, for demonstrational purposes. Support for a larger number or a greater variety of objects would not be difficult because of the automatic data structure conversion functions (see the description of `sib_agent` below for more details).

asibsync can be started up by having all the dependency libraries in `PYTHONPATH` and running the program by executing "`python asibsync`". "`python asibsync help`" gives an overview of possible startup options.

`asibsync` Controller Module

`asibsync` controller contains a simple function that creates an instance of both `ASIAgent` and `SIBAgent` classes, makes them aware of each other and starts both of them by calling their `start` methods, respectively. The main work is then carried out inside the respective `asi_agent` and `sib_agent` modules, which are described in more detail hereafter.

When a user requests a synchronization stop by sending a termination signal to `asibsync`, e.g. by pressing Ctrl-C key combination in the console window, the agents' `stop` methods are called and the whole process ends.

Replacing one or the other of the agents connected to `asibsync` should be possible, as long as the new agent provides a compatible interface and understands the Python structures that are currently used.

`asi_agent`

The `asi_agent` module contains a single class, `ASIAgent`, that takes care of monitoring for and acting upon the updates in relevant ASI objects, and provides an interface for sending updates back to ASI. Marshalling and unmarshalling between JSON strings and Python structures are done with Python 2.6's standard *json* library[21].

As ASI is not currently capable of pushing the updates to clients, polling needs to be resorted to in the monitoring implementation. When the `start` method is invoked, the `ASIAgent` thread is started to run the main loop, where the ASI object to be monitored is retrieved at regular intervals and examined for possible update timestamp changes. In case of an update, the entire object where the change occurred is sent to the `SIBAgent` instance connected to `ASIAgent`.

`ASIAgent` also provides an interface for receiving updates from the other end, in this case `SIBAgent`. The updates received from SIB will be pushed onto an update stack, `pending_sib_updates`, which will be examined at every monitor interval. If the update stack is nonempty and there were no changes on ASI side, `pending_sib_updates` values will be propagated to ASI.

To prevent timing failures, `asi_agent` polling process, implemented in `_poll` func-

[21]http://docs.python.org/library/json.html (referred 5 Apr 2010)

tion, was designed to behave consistently in case of simultaneous or conflicting updates. If ASI was changed, the entire update stack will be discarded. Effectively this means that ASI changes will override any near-simultaneous changes at the other end.

Connection failures and potentially some other transient software are taken into account by handling exceptions in the `ASIAgent` main loop, unless explicitly disabled e.g. for debugging purposes. When an exception is caught from `ASIAgent._poll` or the other functions it calls, it is printed out but otherwise ignored, and the poll loop continues. This ensures that the operation continues even after connection errors. Retrying ASI operations on the next poll cycle should always be possible, since all the operations used are either safe (e.g. HTTP GET in `get_user`) or at least idempotent (HTTP PUT in `update_user`); see the discussion under Subsection 3.3.1.

`asi_agent` polls for changes every `ASIAgent.POLL_INTERVAL` seconds. This polling functionality exists in the `ASIAgent._poll` method. Whenever `_poll` is called, it acquires the update stack lock for the duration of the poll processing to prevent simultaneous updates on the SIB side from taking effect until next `_poll` invocation at the earliest.

Inside `_poll`, the ASI user is retrieved and its timestamp compared to the last stored timestamp, `asi_last_updated`. The actions taken depend on this comparison:

1. If the retrieved timestamp denotes a time after the stored one, an update to SIB is needed. The retrieved user is sent to `sib_agent` and its timestamp stored in `asi_last_updated`. Because `asi_agent` has to win in case of conflicts, the possible values in `pending_sib_updates` queue are discarded.

2. If no update to SIB is needed, check `pending_sib_updates`. If it contains new updated values, send them to ASI and store the `updated_at` value in `asi_last_updated` in order to prevent a useless and possibly harmful SIB update.

Figure 5.3 visualizes the different possible cases in the form of a flowchart.

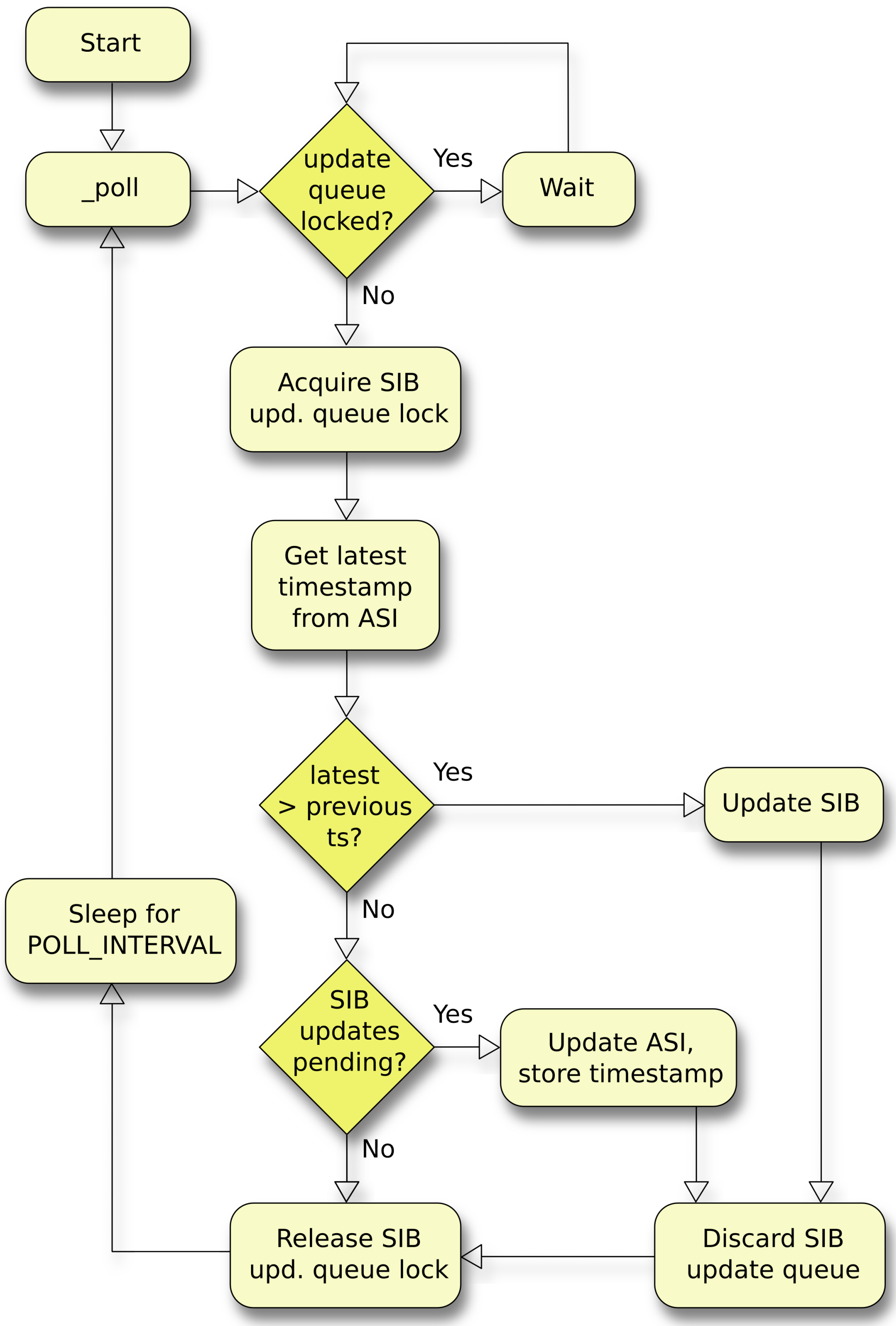


Figure 5.3: `ASIAgent` poll cycle flowchart

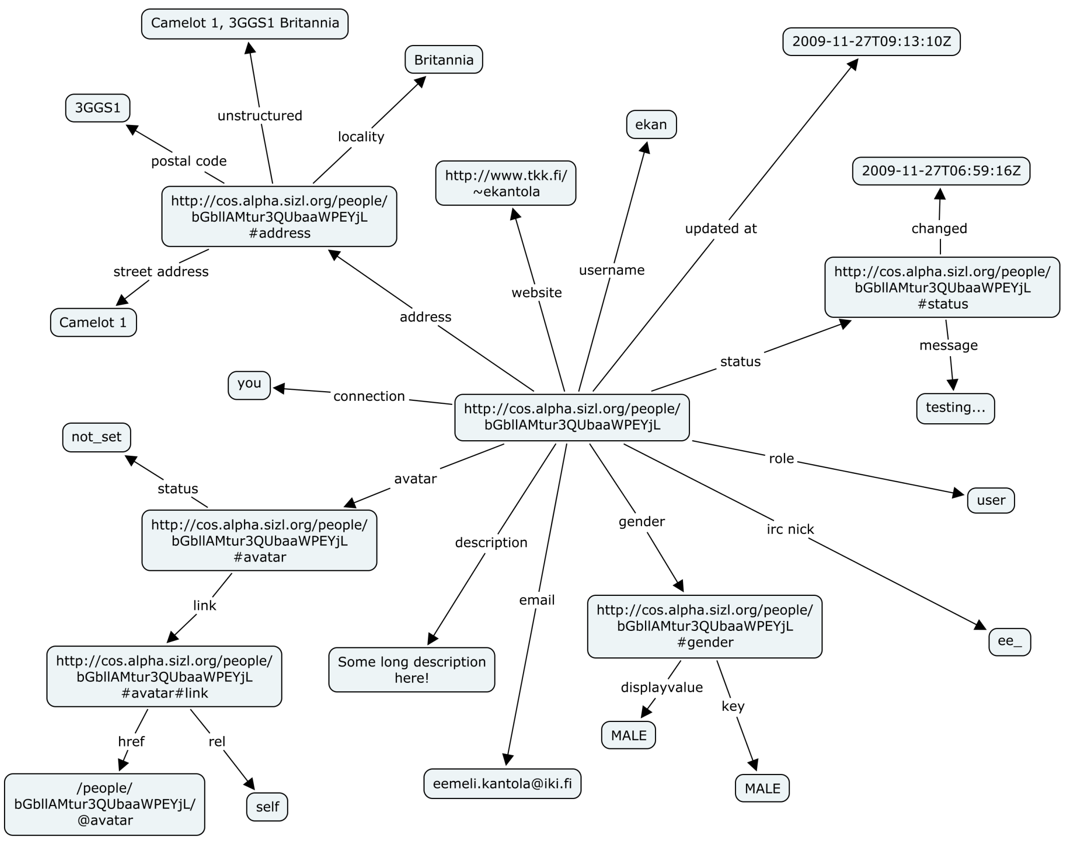

Camelot 1, 3GGS1 Britannia
Britannia
2009-11-27T09:13:10Z
3GGS1
unstructured
locality
ekan
postal code
http://www.tkk.fi/ ~ekantola
2009-11-27T06:59:16Z
http://cos.alpha.sizl.org/people/ bGbllAMtur3QUbaaWPEYjL #address
updated at
changed
street address
username
website
http://cos.alpha.sizl.org/people/ bGbllAMtur3QUbaaWPEYjL #status
Camelot 1
address
message
status
you
connection
http://cos.alpha.sizl.org/people/ bGbllAMtur3QUbaaWPEYjL
testing...
not_set
role
avatar
status
user
http://cos.alpha.sizl.org/people/ bGbllAMtur3QUbaaWPEYjL #avatar
gender
irc nick
description
link
email
http://cos.alpha.sizl.org/people/ bGbllAMtur3QUbaaWPEYjL #gender
ee_
http://cos.alpha.sizl.org/people/ bGbllAMtur3QUbaaWPEYjL #avatar#link
Some long description here!
displayvalue
key
MALE
href
rel
/people/ bGbllAMtur3QUbaaWPEYjL/ @avatar
self
eemeli.kantola@iki.fi
MALE

Figure 5.4: Sample ASI data structure's subset mapped to an RDF graph

sib_agent

The `sib_agent` module contains `SIBAgent` class for subscribing to and processing changes on the SIB side, and also for sending updates from the other end back to SIB, in a similar way that the `ASIAgent` does.

An additional problem in synchronization was converting data to and from the nested Python dictionary structure that is used in `asi_agent`: extracting an RDF ontology as triples, converting the Python struct to a list of RDF instance triples and vice versa. The methods for converting between data structures are also located in the `sib_agent` module. An example of a dataset mapped from ASI to SIB is visualized in Program 3 and Figure 5.4.

SIB provides an interface for subscribing to changes that match a given query. This led to a callback handler that is relatively simple compared to the `ASIAgent`'s added

Program 3 Python structure to RDF ontology and instance mapping sample. This data can be generated from *asibsync* unit tests, `asibsync/tests/sib_agent_test.py`)

Python structure

```
{'address': {'postal_code': '3GGS1',
             'street_address': 'Camelot 1'},
 'avatar': {'link': {'href': 'http://python.org/favicon.ico',
                     'rel': 'self'},
            'status': 'set'},
 'favorite_food': 'eggs',
 'id': '1234',
 'name': 'Eric'}
```

RDF ontology

```
[Triple(uri('https://cos.alpha.sizl.org/people#Person'),
        uri('rdf:type'), uri('rdfs:Class')),
 Triple(uri('https://cos.alpha.sizl.org/people#Address'),
        uri('rdf:type'), uri('rdfs:Class')),
 Triple(uri('https://cos.alpha.sizl.org/people#Avatar'),
        uri('rdf:type'), uri('rdfs:Class')),
 Triple(uri('https://cos.alpha.sizl.org/people#Link'),
        uri('rdf:type'), uri('rdfs:Class'))]
```

RDF instance (truncated)

```
[Triple(uri('https://cos.alpha.sizl.org/people/ID#1234'),
        uri('rdf:type'),
        uri('https://cos.alpha.sizl.org/people#Person')),
 Triple(uri('https://cos.alpha.sizl.org/people/ID#1234'),
        uri('https://cos.alpha.sizl.org/people#name'),
        literal('Eric')),
 Triple(uri('https://cos.alpha.sizl.org/people/ID#1234'),
        uri('https://cos.alpha.sizl.org/people#favorite_food'),
        literal('eggs')),
 Triple(uri('https://cos.alpha.sizl.org/people/ID#1234'),
        uri('https://cos.alpha.sizl.org/people#address'),
        uri('https://cos.alpha.sizl.org/people#1')),
 Triple(uri('https://cos.alpha.sizl.org/people/ID#1234'),
        uri('https://cos.alpha.sizl.org/people#avatar'),
        uri('https://cos.alpha.sizl.org/people#2')),
 Triple(uri('https://cos.alpha.sizl.org/people#1'),
        uri('rdf:type'),
        uri('https://cos.alpha.sizl.org/people#Address')),
 ...]
```

complexity of explicit polling and detecting updated data.

Synchronizing data from SIB back to ASI proved more problematic. The first issue arises from the fact that subscribing to changes in a specific user's data with RDF queries is complicated. Considering the sample ASI dataset mapped as triples in Program 3, it is obvious that when using a simple catch-all wildcard RDF subscription, a single user can not be completely monitored, unless all other changes in SIB are.

The solution implemented here is to initiate multiple RDF subscriptions after ASI data has been synchronized the first time to SIB. In the case of Program 3's sample data structure, a total of four separate RDF subscriptions would be required, one for every non-leaf node (i.e. Python `dict`, or a pair of curly braces in the textual representation) in the original ASI data hierarchy tree. See Program 4 for an outline of what this would look like on the code level. For the actual ASI person, a total number of six subscriptions are needed.

Using WQL, the path query language supported by Smart-M3, could be studied to help reducing the number of subscriptions required for monitoring changes of person structures in SIB.

Program 4 RDF subscriptions that are required for monitoring all possible changes in Program 3's data structure (assuming `#1`, `#2` and `#3` are the SIB identifiers of the person's address, avatar, and avatar's link, respectively)

```
sc = SIBConnection(...)
sc.subscribe(Triple(uri('https://cos.alpha.sizl.org/people/ID#1234'),
             None, None), callback_function)  # Person
sc.subscribe(Triple(uri('https://cos.alpha.sizl.org/people#1'),
             None, None), callback_function)  # address
sc.subscribe(Triple(uri('https://cos.alpha.sizl.org/people#2'),
             None, None), callback_function)  # avatar
sc.subscribe(Triple(uri('https://cos.alpha.sizl.org/people#3'),
             None, None), callback_function)  # avatar_link
```

Another problem was mapping the graph data structure from Smart-M3 back to ASI's hierarchical format. Converting the graph back to a tree structure is doable in this case because the structure was originally converted from ASI. However, traversing the graph back to the Person object is needed, which involves extra logic and queries to deduce the original ASI data structure.

For instance, assume that an agent has subscribed to changes in a person's avatar link with the query shown in Program 5. When the specific avatar link is updated, following steps will take place:

1. `callback_function` is called with the updated triple. Inside the callback, a subject URI that looks like person structure identifier (ending in `people#3`) is detected.
2. Triples with object `people#3` are searched, and `people#2` is found. The matching triple's predicate is investigated and the type 'link' is noted.
3. Triples with object `people#2` are searched, and `people/ID#1234` is found. This is the hierarchy root. Matching triple's predicate is investigated and the type 'avatar' is noted.
4. Now we know that the triple updated was person ID 1234's avatar's link, and the corresponding ASI update request can be crafted.

Similarly to `ASIAgent`, `SIBAgent` offers a method for receiving updates from the other end and transmitting them to SIB. The data, however, is stored as RDF/RDFS in SIB, which means that the ontology may not exist on the SIB side prior to updating data. This case is handled by the receive method on the first run by generating a matching RDFS ontology from the received structure and updating it to SIB if needed.

Program 5 Sample data for mapping from RDF triples to ASI structure

Query for subscribing to avatar's link (see RDF below)

```
sc = SIBConnection(...)
sc.subscribe(Triple(uri('https://cos.alpha.sizl.org/people#3'),
                    None, None), callback_function)
```

RDF structure as triples

```
[...
 Triple(uri('https://cos.alpha.sizl.org/people#2'),
        uri('rdf:type'),
        uri('https://cos.alpha.sizl.org/people#Avatar')),
 Triple(uri('https://cos.alpha.sizl.org/people#2'),
        uri('https://cos.alpha.sizl.org/people#link'),
        uri('https://cos.alpha.sizl.org/people#3')),
 Triple(uri('https://cos.alpha.sizl.org/people#2'),
        uri('https://cos.alpha.sizl.org/people#status'),
        literal('set')),
 Triple(uri('https://cos.alpha.sizl.org/people#3'),
        uri('rdf:type'),
        uri('https://cos.alpha.sizl.org/people#Link')),
 Triple(uri('https://cos.alpha.sizl.org/people#3'),
        uri('https://cos.alpha.sizl.org/people#href'),
        literal('http://python.org/favicon.ico')),
 Triple(uri('https://cos.alpha.sizl.org/people#3'),
        uri('https://cos.alpha.sizl.org/people#rel'),
        literal('Eric'))]
```

Update statement. Note: only object part is updated:

```
sc.update(Triple(uri('https://cos.alpha.sizl.org/people#3'),
                 uri('https://cos.alpha.sizl.org/people#href'),
                 literal('http://python.org/favicon.ico')),
          Triple(uri('https://cos.alpha.sizl.org/people#3'),
                 uri('https://cos.alpha.sizl.org/people#href'),
                 literal('http://kassi.sizl.org/favicon.ico')))
```

Generated Python structure:

```
{'avatar': {'link': {'href': 'Eric',
                     'rel': 'http://kassi.sizl.org/favicon.ico''}},
 'id': '1234'}
```

Chapter 6

Analysis

In this Chapter the software implementation is evaluated in light of the original goals, namely synchronizing between ASI and Smart-M3, evaluating Smart-M3's knowledge processor library, and a publish/subscribe mediator for a traditional web service.

6.1 Synchronizing Between ASI and Smart-M3

This Section first discusses the actual synchronization implementation for synchronizing in two directions, and then evaluates the software's fault tolerance and performance.

6.1.1 ASI to Smart-M3

ASI contains existing user data to be utilized in synchronizing to Smart-M3. This was taken advantage of when implementing the data structure and content synchronization functions, and consequently synchronizing was shown to work well at least in the case of a relatively simple user data structure.

Currently `asi_agent` is limited to synchronizing ASI's person data structures only. Other types of data can also be supported with small changes to the code, but for more general support `asi_agent` should be enhanced to parse the synchronization options and definitions of the data structures from e.g. a configuration file.

Also, as the whole ASI person is retrieved and pushed to SIB every time there was an update, possible non-conflicting updates in SIB are discarded. Solving the problem would need more sophisticated algorithms for tracking changes on ASI's side and detecting ASI-SIB conflicts.

Additionally it should be noted that `sib_agent` does not remove old values from SIB but they are left there as a kind of a history. However, the history will be unsorted and the most recent versions of ASI object properties remain indistinguishable from the historical values.

6.1.2 Smart-M3 to ASI

Synchronizing in the other direction proved more problematic, even in case of data originally synchronized from ASI. This was doable, however, with two additional measures:

- by storing references to the triples that represent different nodes in the original tree, and subscribing to changes to them; and
- by having the RDF data structure contain enough information to be mappable back to ASI's tree structure, and traversing the RDF graph back to the original tree structure's root.

Synchronizing from Smart-M3 (or, more generally, any other source of free-form, graph-structured information) without having the data mapped first from ASI would require manual mapping in general case. In theory, it could be possible to automate the process if the data structures to be mapped were specified and properly formatted.

In the current implementation, performance problems will arise with deeper structures because traversing the whole graph path back to the root node is needed. This could be circumvented by introducing an indexing directory, either stored as RDF in Smart-M3 or cached locally in *asibsync*, to map the descendant node URIs directly to the original tree root in graph.

The current *asibsync* controller implementation has the data type hardcoded to handle ASI person mapping only. The type could be easily changed in code, however. In general case, support of mapping information from Smart-M3 to ASI could be added by enhancing the current URI parsing based implementation to take other ASI cases into account. Investigating all the relevant use cases would be needed, and it should be noted that this would potentially involve in syntactic improvements to the current URI formatting conventions.

6.1.3 Fault Tolerance

asibsync's fault tolerance was analyzed by first running the program in normal conditions, and then artificially introducing network errors to SIB and ASI before and after starting up to find out how *asibsync* can recover from them. The test environment was the same as described in Section 5.2, with the ASI to SIB synchronization having already taken place at least once to ensure that ontology exists in SIB.

The induced network error in ASI's case was entirely disconnecting the test host from the network by pulling off the Ethernet cable. On the other hand, the procedure for disconnecting SIB was to temporarily shut down the `sib-tcp` connector program, as the SIB was easily controllable in this test setup.

After introducing connection errors and recovering from them, a verification test was conducted to ensure proper functioning after testing. The test procedure was following:

1. Update user's email address in SIB
2. Verify that the updated address was propagated to ASI
3. Change the email address back to the original value in ASI
4. Verify that the original address was propagated back to SIB

Normal Conditions

Initially, synchronization was started and carried out by running *asibsync* without added network problems. The output produced can be seen in Program 6 listing. First *asibsync* successfully sets up `ASIAgent` and then `SIBAgent` that open ASI and SIB connections, respectively. After that, the initial synchronization is carried out, RDF subscriptions are set up and `ASIAgent` polling starts.

When running under normal conditions, the verification test passed without problems.

Program 6 *asibsync* behavior when started with "`python asibsync -de`" and run under normal conditions

```
[2010-04-19 01:43:35.810962] ASIAgent: Debug mode enabled
[2010-04-19 01:43:37.669614] SIBAgent: Debug mode enabled
[2010-04-19 01:43:37.669821] kpwrapper: Preconfigured discovery using
config from /Users/ekan/.kprc
[2010-04-19 01:43:37.670010] kpwrapper: Got params: ('x', ('TCP',
('192.168.216.10', 10010)))
Press enter to request sync stop.
[2010-04-19 01:43:37.670510] ASIAgent: Configuration has been done,
starting.
[2010-04-19 01:43:37.792370] ASIAgent: Updating SIB
(2010-04-18T21:49:59Z > None)
[2010-04-19 01:43:37.792425] SIBAgent: receive called
[2010-04-19 01:43:37.792481] SIBAgent: Generating ontology
[2010-04-19 01:43:37.871707] SIBAgent: Ontology seems to be up to date
[2010-04-19 01:43:37.871748] SIBAgent.generate_ontology processed in
0.079274 s
[2010-04-19 01:43:37.871823] SIBAgent: Received {u'username': ... }
[2010-04-19 01:43:37.872860] SIBAgent: Inserting [Triple(...), ...)]
[2010-04-19 01:43:37.917712] SIBAgent: Subscribing to changes
[2010-04-19 01:43:37.993242] SIBAgent: Subscribed to Triple(uri('http:
//cos.alpha.sizl.org/people/ID#bGbllAMtur3QUbaaWPEYjL'), None, None)
[2010-04-19 01:43:38.018801] SIBAgent: Subscribed to Triple(uri('http:
//cos.alpha.sizl.org/people#00fee017-482a-4cd4-a97f-958aa9bb0d5d'),
None, None)
...
[2010-04-19 01:43:38.178269] SIBAgent.subscribe processed in
0.260573 s
[2010-04-19 01:43:38.178323] SIBAgent.receive processed in 0.385904 s
[2010-04-19 01:43:38.178356] ASIAgent._poll processed in 0.507799 s
[2010-04-19 01:43:45.030070] ASIAgent: No need to update SIB
(2010-04-18T21:49:59Z <= 2010-04-18T21:49:59Z)
[2010-04-19 01:43:45.030127] ASIAgent._poll processed in 1.720626 s
[2010-04-19 01:43:52.320692] ASIAgent: No need to update SIB
(2010-04-18T21:49:59Z <= 2010-04-18T21:49:59Z)
[2010-04-19 01:43:52.320746] ASIAgent._poll processed in 2.291834 s
...
```

Connection Down Before Startup

In the second phase of tests, connection first to SIB and then to ASI was intentionally broken before starting up *asibsync*, and then restored after ten seconds of running unconnected. Programs 7 and 8 list the relevant *asibsync* output in these cases. These listings show that the connection errors are handled gracefully, with the program being able to recover and continue operating normally after the connections start functioning properly.

With *asibsync* running after recovering from the initial connection problem, the verification test passed without problems.

Connection Error While Running

Lastly, connections were broken after the startup phase and at least one initial synchronization cycle had successfully taken place. *asibsync* outputs from the tests can be seen in Programs 9 and 10. In SIB's case, the subscribe transactions crash with a `KeyError` from the *python-kp* library.

After having recovered from the ASI connection error, the verification test completed without problems. However, after SIB connection error updating the email address in SIB did not produce any effect on ASI side, indicating an unrecoverable error.

Summary of fault tolerance

To summarize the tests that were conducted, the synchronization agent can recover from many communication failures such as ASI connection errors that occur after starting up the agent is finished, and also many other transient errors in `ASIAgent._poll` or the methods that it calls directly and indirectly, namely `SIBAgent.receive` and other calls within it. Non-recurring process failures are also likely to be recoverable to a degree in the methods mentioned, although verifying this would require more studies.

However, currently SIB connection errors are not handled so well, as can be seen in Program 9 output. The subscribe transactions rely on the TCP socket to be connected all the time, and if not, the subscriptions stop functioning even if the connection could later be restored. This is a shortcoming of the underlying Smart-M3 *python-kp* library, and would require either support for automatic reconnecting on

Program 7 *asibsync* behavior when started with "`python asibsync -de`" and the connection to SIB is down for a while at startup

```
[2010-04-19 01:50:33.316625] ASIAgent: Debug mode enabled
[2010-04-19 01:50:33.431941] SIBAgent: Debug mode enabled
[2010-04-19 01:50:33.432172] kpwrapper: Preconfigured discovery using
config from /Users/ekan/.kprc
[2010-04-19 01:50:33.432450] kpwrapper: Got params: ('x', ('TCP',
('192.168.216.10', 10010)))
[2010-04-19 01:50:33.432791] ASIAgent: Configuration has been done,
starting.
Press enter to request sync stop.
[2010-04-19 01:50:33.558423] ASIAgent: Updating SIB
(2010-04-18T21:49:59Z > None)
[2010-04-19 01:50:33.558479] SIBAgent: receive called
[2010-04-19 01:50:33.558503] SIBAgent: Generating ontology
[2010-04-19 01:50:33.559148] ASIAgent: Caught exception: [Errno 61]
Connection refused
[2010-04-19 01:50:38.655534] ASIAgent: Updating SIB
(2010-04-18T21:49:59Z > None)
[2010-04-19 01:50:38.655591] SIBAgent: receive called
[2010-04-19 01:50:38.655616] SIBAgent: Generating ontology
[2010-04-19 01:50:38.656236] ASIAgent: Caught exception: [Errno 61]
Connection refused
[2010-04-19 01:50:43.754045] ASIAgent: Updating SIB
(2010-04-18T21:49:59Z > None)
[2010-04-19 01:50:43.754104] SIBAgent: receive called
[2010-04-19 01:50:43.754129] SIBAgent: Generating ontology
[2010-04-19 01:50:43.988245] SIBAgent: Ontology seems to be up to date
[2010-04-19 01:50:43.988426] SIBAgent.generate_ontology processed in
0.234302 s
[2010-04-19 01:50:43.993895] SIBAgent: Received {u'username': ... }
...
```

Program 8 *asibsync* behavior when started with “`python asibsync -de`” and the connection to ASI is down for a while at startup

```
[2010-04-19 02:02:16.608501] ASIAgent: Debug mode enabled
[2010-04-19 02:02:16.722711] ASIAgent: Caught exception: <urlopen
error [Errno 8] nodename nor servname provided, or not known>
[2010-04-19 02:02:21.823955] ASIAgent: Caught exception: <urlopen
error [Errno 8] nodename nor servname provided, or not known>
[2010-04-19 02:02:27.255409] SIBAgent: Debug mode enabled
[2010-04-19 02:02:27.255630] kpwrapper: Preconfigured discovery using
config from /Users/ekan/.kprc
[2010-04-19 02:02:27.255808] kpwrapper: Got params: ('x', ('TCP',
('192.168.216.10', 10010)))
[2010-04-19 02:02:27.256220] ASIAgent: Configuration has been done,
starting.
Press enter to request sync stop.
...
```

Program 9 *asibsync* behavior when started with “`python asibsync -de`” and the connection to SIB breaks after starting up

```
...
[2010-04-19 02:43:46.583516] ASIAgent: No need to update SIB
(2010-04-18T21:49:59Z <= 2010-04-18T21:49:59Z)
[2010-04-19 02:43:46.583765] ASIAgent._poll processed in 0.096184 s
Exception in thread Thread-2:
Traceback (most recent call last):
  File "/System/Library/Frameworks/Python.framework/Versions/2.6/lib/
python2.6/threading.py", line 522, in __bootstrap_inner
    self.run()
  File "/Users/ekan/workspace/asibsync/eggs/smart_m3_pythonKP-0.9.0-
py2.6.egg/smart_m3/Node.py", line 1155, in run
    if msg["transaction_type"] == "SUBSCRIBE":
KeyError: 'transaction_type'

Exception in thread Thread-3:
...
[2010-04-19 02:43:55.437505] ASIAgent: No need to update SIB
(2010-04-18T21:49:59Z <= 2010-04-18T21:49:59Z)
[2010-04-19 02:43:55.437555] ASIAgent._poll processed in 3.854900 s
...
```

Program 10 *asibsync* behavior when started with "`python asibsync -de`" and the connection to ASI breaks after starting up

```
...
[2010-04-19 02:46:51.274109] ASIAgent: No need to update SIB
(2010-04-18T21:49:59Z <= 2010-04-18T21:49:59Z)
[2010-04-19 02:46:51.274167] ASIAgent._poll processed in 1.120647 s
[2010-04-19 02:46:56.309477] ASIAgent: Caught exception: <urlopen
error [Errno 8] nodename nor servname provided, or not known>
[2010-04-19 02:47:01.310633] ASIAgent: Caught exception: <urlopen
error [Errno 8] nodename nor servname provided, or not known>
[2010-04-19 02:47:08.030070] ASIAgent: No need to update SIB
(2010-04-18T21:49:59Z <= 2010-04-18T21:49:59Z)
[2010-04-19 02:47:08.030127] ASIAgent._poll processed in 1.720626 s
...
```

python-kp's side, or somehow working around the issue on *asibsync* side by monitoring the subscribe transaction threads and reopening them when this becomes possible.

Studying timing failures was not concentrated on, as the *asibsync* architectural design described in Subsection 5.3.3 renders them largely irrelevant. Arbitrary failure recovery was not taken into account in design nor studies.

6.1.4 Performance

Synchronization performance was tested using a single person object from ASI. The same test person from ASI was used throughout the process. Platform setup was the same as described in Section 5.2. A single test run was carried out by running "`python asibsync -de`" and observing the runtimes of interesting functions from the debug output.

Before starting the actual measured tests, the test were dry-run three times to ensure operating system had cached program code and libraries to memory as far as possible, thus minimizing the effects of extra filesystem operations. The test was repeated ten times for both cases to obtain rough estimates of where the synchronization bottlenecks could be.

There were two different scenarios that were tested: empty database on SIB side, which was achieved by deleting the database file and restarting SIB before the test; and SIB with a database to which the ASI person had been already synchronized

Table 6.1: Summary of the performance measurements for ten repetitions. First column displays the function name to be measured, the second and third contain their average running times with standard deviations in parenthesis for empty and nonempty databases, respectively.

	Function	Empty avg (stdev)	Nonempty avg (stdev)
t_p	`_poll`	1.3 s (0.090 s)	0.92 s (0.062 s)
t_r	`SIBAgent.receive`	0.84 s (0.066 s)	0.51 s (0.066 s)
t_g	`generate_ontology`	0.35 s (0.029 s)	0.084 s (0.0098 s)
t'_p	$t_p - t_r - t_g$	0.087 s (0.032 s)	0.33 s (0.015 s)
t'_r	$t_r - t_g$	0.49 s (0.055 s)	0.42 s (0.063 s)

once.

Three different cases were measured:

1. `SIBAgent.generate_ontology`
2. `ASIAgent._poll`
3. `SIBAgent.receive`

Before the first actual measurement run, the test was run three times to minimize the effect of possible disk usage.

Table 6.1 summarizes the results of measured runtimes of `_poll`, `generate_ontology`, and `SIBAgent.receive`. The full results are listed in Appendix A. Additionally `ASIAgent._poll` internally calls `generate_ontology` and `SIBAgent.receive`, hence the added fourth row in the table with `_poll` runtime excluding `generate_ontology` and `SIBAgent.receive`.

Overall, the measured times were highly predictable across different test runs, as can be noted by examining the standard deviations which were an order of magnitude below the respective averages measured (not including t'_p and t'_r).

`SIBAgent.generate_ontology` executes quickly when there is no need to update the ontology. In this case, `generate_ontology` performs six queries in SIB, but no updates. On the other hand, when creating ontology was needed, the runtime average is several times higher than when no update was needed. Creating involves in inserting six RDF triples to SIB. This suggests that the insert operations are significantly more expensive than queries.

The initial conclusion of the tests is that updating SIB (i.e. creating triples) is slow. While confirming this would need more in-depth studies, these preliminary results indicate that Smart-M3 performance may become a bottleneck even with a few near-simultaneous updates.

6.1.5 Further Notes

Synchronizing between two endpoints, as is the case here, is fairly straightforward when one end can be defined as the primary authority that overrides in case of conflicting updates. However, it is noteworthy that should *asibsync* be modified for synchronizing to Smart-M3 or equivalent from several data sources, then more sophisticated algorithms need to be introduced.

The different types of reasonable modifications would be as follows.

- Changing from ASI to a different Web Service would involve in obtaining or writing a library for interfacing with the new Web Service, and implementing a new agent, say `new_web_service_agent`, to connect to `sib_agent`. This `new_web_service_agent` would need to be able to cope with the JSON-like Python data structure used in `sib_agent`'s interface. The *asilib* implementation could be used as an example implementation, however, or even extended to support the new Web Service at least if a RESTful service similar to ASI would be in question.
- Changing from Smart-M3 to a different Smart Space would need a library for managing the other Smart Space and agent, say `new_smart_space_wrapper`, to connect to `asi_agent`. `new_smart_space_wrapper` should, if possible, conform to the *kpwrapper* library interface, such as `Triple` abstraction and transactions, to avoid duplicating the work already done.
- When changing both ASI and SIB, more work would be involved. Although the main `asibsync` module and possibly a large part of *kpwrapper* could be reused, at least most of `asi_agent` would need to be replaced. In any case, the existing implementation could be used as an example for supporting other endpoints, if not extended and generalized at least to some degree.

Another point of improvement could be enhancing the RDF data model to better support hierarchical data with similarly named nodes. This is not a problem with the ASI person structure used in the current implementation, but could be an issue with some different data types. For instance, `link` could be used for other links in addition to `avatar`'s, such as a new fictitious `homepage` property, in which case both `avatar`'s and `homepage`'s `link`s would result in the exactly same RDF class description,

```
Triple(uri('https://cos.alpha.sizl.org/people#Link'),
       uri('rdf:type'), uri('rdfs:Class'))]
```

One way to alleviate this problem would be to construct the RDF class subject URIs to contain the full hierarchy information:

```
Triple(uri('https://cos.alpha.sizl.org/people#Avatar/Link'),
       uri('rdf:type'), uri('rdfs:Class'))]
Triple(uri('https://cos.alpha.sizl.org/people#Homepage/Link'),
       uri('rdf:type'), uri('rdfs:Class'))]
```

6.2 Smart-M3's Knowledge Processor Library

One of the original goals of this Thesis was using and evaluating Smart-M3's knowledge processor library, *python-kp*, from a developer's point of view. This Section summarizes the qualitative analysis of the library.

The knowledge processor library was found partly unstable, particularly in case of unexpected inputs. This lead to a number of workarounds in the *kpwrapper* library, such as not actually closing the underlying SIB connection when it would result in a program crash. Also for some reason the SIB daemon (`sibd`) always crashes in the test setup if `sib_agent` uses triple URIs starting with `https://` instead of `http://`. A workaround exists in `asibsync.__init__` where the RDF base URI is initially set.

The library interface was found difficult to use for the common case concerning software developed for this Thesis. Thus a simplifying wrapper library was written on top of the knowledge processor library. Program 11 demonstrates how *kpwrapper* simplifies subscription handling for the programmer compared to using the underlying *python-kp* library directly. The required amount of program code is significantly smaller using *kpwrapper*.

Program 11 Comparing *python-kp* and *kpwrapper* with opening and managing subscriptions. Both program code snippets are roughly functionally equivalent.

(a) Subscribing to changes with *python-kp*

```
class CallbackHandler:
    def handle(self, added, removed):
        try:
            for triple in added:
                do_something_with(added[0])  # subject
                ...
        except Exception, e:
            # Don’t let possible exceptions crash python-kp
            print(’CallbackHandler.handle: Caught exception %s’ % e)

kp = ParticipantNode(’Node’)
ss = (’X’, (TCPConnector, (’127.0.0.1’, 10010)))
if not kp.join(handle):
    raise Exception(’Join failed’)

t = ((’https://cos.alpha.sizl.org/people/ID#1234’, None, None),
     ’uri’, ’literal’)
sub_tx = kp.CreateSubscribeTransaction(ss)
sub_tx.subscribe_rdf([t], CallbackHandler(), True)

raw_input(’Press enter to stop listening to changes’)
sub_tx.close()
```

(b) Subscribing to changes with *kpwrapper*

```
def callback_handler(added, removed):
    for triple in added:
        do_something_with(added.subject)
        ...

with SIBConnection(method=’preconfigured’) as sc:
    t = Triple(’https://cos.alpha.sizl.org/people/ID#1234’, None, None)
    sc.subscribe(t, callback_handler)
    raw_input(’Press enter to stop listening to changes’)
```

Other problems as a developer arose from the relatively complicated nature of the

Semantic Web technologies, at least in Smart-M3's context. *kpwrapper* hides the libraries' technical complexity to some degree, but figuring out the exact RDF structure for different kinds of mapped data could provide problematic for typical web developers as the technologies are not common in web application or Web Service development.

Authentication and authorization are not currently taken into account in SIB or *python-kp*. Taking security into account would be required in order to facilitate the adoption of Smart-M3 in real world applications. However, the problem is mitigated if *asibsync* and SIB are to run on the same host with firewalling and other standard security measures in place.

6.3 Non-Intrusive Publish/Subscribe Mediator for ASI

This Section briefly evaluates *asibsync*'s ability to act as a non-intrusive pubsub mediator for ASI, and discusses the potential problems.

With the current polling interval of five seconds, thousands of subscribers implying hundreds of additional requests per second may not be a sustainable solution from the server's scalability point of view. At least further performance studies and polling interval optimization should be conducted in order to determine ASI's scalability properties.

For subscriptions with infrequent updates, utilizing server push techniques such as long polling on ASI's side should be considered. For providing high scalability with a large number of pending requests, employing a solution like the *Juggernaut*[1] plugin for Ruby on Rails could be considered.

These notes apply more generally to any equivalent service to be adapted to support publish/subscribe under similar update frequency requirements. Thus adapting services incapable of scalable server push could not be considered non-intrusive when higher loads are to be expected.

[1]http://juggernaut.rubyforge.org (referred 30 Apr 2010)

6.4 Overcoming Distributed System Challenges in *asibsync*

There exist several challenges for distributed systems, as introduced in Subsection 2.1.2. This Section revisits the challenges and summarizes the results of this Chapter.

Heterogeneity is one of the challenges that Smart-M3 in combination with *asibsync* are designed to mask. By synchronizing ASI data structures via asibsync to Smart-M3, it becomes possible to map data to structures imported from other services.

Openness and the added complexity of having ASI's and SIB's interfaces publicly available for use by external developers is less of a problem on ASI's side because of the RESTful architectural choices. Smart-M3 with the Semantic Web technologies provide a framework designed for extensions. On the other hand, Smart-M3 is facing challenges in providing a simple, clean and working interface to external developers.

Security between ASI and the *asibsync* is ensured by HTTPS and HTTP cookies with a hashed session id. This is secure enough for most practical purposes. Smart-M3, on the other hand, does not currently incorporate any authentication mechanisms and should not be used for applications with security requirements.

Scalability of SIB seems problematic, considering the performance test results. ASI scalability was found sufficient for this Thesis' purposes, but having to resort to polling might pose problems. In any case more studies would be required, as evaluating the system's scalability was not the main focus of this Thesis.

Failure handling of *asibsync* can mask connection-related and possibly many transient software failures, except regarding SIB subscriptions after starting up. Updates on SIB side will not be propagated to ASI after a SIB connection problem or crash, and will currently require *asibsync* restart.

Concurrency, or in this case concurrent and conflicting updates from both endpoints, has been handled in *asibsync* for the two-endpoint case studied in this Thesis. For future applications and aggregating data in Smart-M3 from several other services, the algorithms should be revised and possibly improved.

Transparency is promoted by *asibsync* by hiding a part of a connected Web Service's, i.e. ASI's, implementation details. This is enabled by Smart-M3 for storing information and facilitating the combining of ASI's data with other information in the future.

Chapter 7

Conclusions

7.1 Summary About the Results

The three primary goals for this Thesis were to connect ASI REST interface with Smart-M3 for synchronizing data, to use Smart-M3's knowledge processor library in a realistic setup, and to implement and evaluate a publish/subscribe mediator for a traditional web service.

The first goal, synchronizing between ASI and Smart-M3, was achieved. Synchronizing from ASI to Smart-M3 works as expected, with no limitations in theory and little so in practice. The synchronization agent can recover from several connection problems, with some limitations regarding Smart-M3's knowledge processor library. The agent could also be enhanced to recover from transient software crashes. However, synchronizing from Smart-M3 to ASI is more problematic because of the challenges involved in monitoring updates. This Thesis shows that it is possible at least in a single ASI person's case, but more studies would be needed to confirm if this is possible and practically feasible with more data and complex datasets.

The second goal of using Smart-M3's knowledge processor library for producing a working application was fully achieved. However, as the library interface was found difficult to use for the common case concerning software developed for this Thesis, a simplifying wrapper library was written on top of the knowledge processor library. This also has the additional advantage of reducing the number of changes needed to replace Smart-M3 as the other endpoint, should that be necessary.

Thirdly, a non-intrusive publish/subscribe mediator for a traditional RESTful web service, ASI, was implemented. This mediator was used in the synchronization agent to subscribe to and monitor changes in ASI's end. Polling at regular intervals is needed for monitoring changes, though, which may affect ASI's performance if updates need to be checked frequently by many simultaneous subscriptions from different sources.

7.1.1 Synchronization Challenges Revisited

An evaluation of how the challenges presented in Section 1.1 were handled in this Thesis follows.

First, the incompatible data format challenge was overcome by writing conversion functions and devising a special URI convention for storing data as RDF triples. There are still issues to be considered when mapping between a hierarchical structure to a graph, but overall this seems achievable even on a general level as long as tree-like structures are involved.

Next, detecting data changes in ASI was made possible by polling to simulate subscriptions and further to implement a publish/subscribe mediator. Loading and updating only relevant changes would need further studies, however.

Thirdly, data synchronization management was implemented inside the `asi_agent` module's polling cycle. This was relatively straightforward in the current setup with two synchronization endpoints, but could become a greater challenge when synchronization with multiple endpoints is required.

The last synchronization challenge, scalability, was analyzed but not yet solved at a sufficient level for large-scale deployments. The solutions suggested include introducing better publish/subscribe support for ASI and performance improvements for Smart-M3, especially regarding data updates.

7.2 Problems Encountered

Two major issues arose during constructing the software that need further attention. Firstly, Smart-M3 is relatively difficult to use in an integrator's point of view. The Python interface provided is not very intuitive for a typical Python programmer, which was in this case circumvented by writing a wrapper library. Secondly, the RDF-based data model is not well-known to typical web developers and, although powerful and flexible, would require extra learning effort for those used to the hierarchical structures commonly used in internet services. These problems are likely to hinder Smart-M3's adoption for data aggregation platform. Improved developer tool support would be needed to help overcome the steep learning curve.

A few other issues encountered were Smart-M3's performance and privacy protection, which were found lacking. For instance, updating Smart-M3 SIB was found to last much longer than would be expected in the preliminary studies. This implies that performance could become a problem even in case of a few simultaneous data updating agents. The performance properties on the SIB's end should be examined thoroughly before Smart-M3 were to be relied on in more serious use.

As for the security, starting SIB will open a TCP port accessible for anyone if a firewall has not been properly set up to prevent requests from outside. In a multiuser environment, this would be a more difficult problem since firewalling does not prevent other users accessing the SIB. The privacy protection issues would need further attention even regarding strictly personal Smart Spaces, if aggregating sensitive personal data.

ASI's suitability for synchronizing data was generally found sufficient, as no major problems were encountered during development and testing. However, synchronizing could be made more efficient if the interface could provide a way to subscribe to changes instead of resorting to polling. Also, it could also be possible to only receive the parts of data that were updated instead of a complete data object, which would decrease the traffic between ASI and the synchronization agent. These may become issues in case of a large number of synchronization agents.

7.3 Future Research and Applications

Synchronizing between Smart-M3 and ASI raises numerous possible directions for future research.

As noted earlier in this Chapter, ASI's and Smart-M3's scalability should be studied in more detail. This includes considering server push and publish/subscribe support for ASI, and Smart-M3 performance especially with data updates. Furthermore, Smart-M3's security and stability should be studied and improved before wider deployment.

In order to facilitate Smart-M3's adoption among developers, developing an interface for the knowledge processor agents to enable quick integration to external services is essential. Currently generating and mapping the ontology requires manual work with technical insight into the mapping procedures, and programming interface to the external services needs to be constructed separately.

As for the applications, one possible use case would be combining friend information to a personal Smart Space from other social networking services, in addition to ASI. This friend information could be then propagated back to the other participant services, keeping information up to date across the relevant social networks. Synchronizing multiple services via Smart-M3 would need to be studied in more detail, however.

Showing news concerning the places of my friends' residence is another noteworthy use case. The relevant locations would first be gathered and combined to a personal Smart Space, and then used for generating a list of interesting places. Then the news articles concerning these locations could be filtered, either by using location metadata provided along the news items, or using e.g. data mining techniques.

The numeric and semantic accuracy of a user's location information could be increased by weighting or prioritizing information combined from e.g. a user's positioning from a GPS-enabled handheld device, textual location information from a social networking service, and the geolocation information of the user's last assigned internet address.

More generally, Smart Spaces could be used for aggregating and taking advantage of users' context information. A practical example would be combining the restaurant menus near a certain location, its accuracy and the user's history to produce suggestions for lunch restaurants, a scenario suggested by Hälikkä (2010).

Bibliography

Alani H., Chandler P., Hall W., O'Hara K., Shadbolt N., & Szomszor M. 2008. Building a Pragmatic Semantic Web. *IEEE Intelligent Systems*, **23**(3), 61–68.

Alonso G., Casati F., Kuno H., & Machiraju V. 2004. *Web services: concepts, architectures and applications.* Springer Verlag.

Ankolekar A., Krötzsch M., Tran T., & Vrandecic D. 2010. The two cultures: Mashing up Web 2.0 and the Semantic Web. *Page 834 of: Proceedings of the 16th international conference on World Wide Web.* ACM.

Battle R., & Benson E. 2008. Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST). *Web Semantics: Science, Services and Agents on the World Wide Web*, **6**(1), 61 – 69. Semantic Web and Web 2.0.

Beckett D., & McBride B. 2004. *RDF/XML Syntax Specification (Revised), W3C Recommendation 10 February 2004.* Tech. rept. World Wide Web Consortium.

Berners-Lee T. 1996. WWW: Past, Present, and Future. *IEEE Computer*, **29**(10), 69–77.

Berners-Lee T., Masinter L., & McCahill M. 1994. *RFC 1738: Uniform Resource Locators (URL).* Tech. rept. IETF.

Berners-Lee T., Hendler J., & Lassila O. 2001. The semantic web. *Scientific American*, **284**(5), 34–43.

Berners-Lee T., Fielding R., & Masinter L. 2005. *RFC 3986: Uniform resource identifier (uri): Generic syntax.* Tech. rept. The Internet Society.

Bray T., Paoli J., Sperberg-McQueen C. M., Maler E., & Yergeay F. 2008 (November). *Extensible Markup Language (XML) 1.0 (Fifth Edition).* Tech. rept. World Wide Web Consortium. Recommendation REC-xml-20081126.

Castro M., & Liskov B. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, **20**(4), 398–461.

Coulouris G.F., Dollimore J., & Kindberg T. 2005. *Distributed systems: concepts and design.* Addison-Wesley Longman.

Crockford D. 2006. *RFC 4627: Javascript object notation.* Tech. rept. The Internet Society.

Eugster P.T., Felber P.A., Guerraoui R., & Kermarrec A.M. 2003. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, **35**(2), 131.

Fidge C.J. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Pages 56–66 of: Proceedings of the 11th Australian Computer Science Conference*, vol. 10.

Fielding R. 2000. *Architectural styles and the design of network-based software architectures.* Ph.D. thesis, University of California, Irvine.

Fielding R., Gettys J., Mogul J., Frystyk H., Masinter L., Leach P., & Berners-Lee T. 1999. *RFC 2616: Hypertext Transfer Protocol–HTTP/1.1.* Tech. rept. The Internet Society.

Gamma E., Helm R., Johnson R., & Vlissides J.M. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

Ghosh S. 2006. *Distributed systems: an algorithmic approach.* CRC press.

Google. 2009 (March). *OpenSocial Specification 1.0.* Tech. rept. OpenSocial and Gadgets Specification Group.

Hadzilacos V., & Toueg S. 1994. *A Modular Approach to Fault-tolerant Broadcasts and Related Problems.* Tech. rept. Dept of Computer Science, University of Toronto.

Honkola J., Laine H., Brown R., & Oliver I. 2009. Cross-Domain Interoperability: a Case Study. *2nd Russian Conference on Smart Spaces, ruSMART 2009.*

Hälikkä H. 2010. *Developing context-aware mobile widgets with S60 Web Runtime.* M.Sc.Tech. thesis, Aalto University.

Lamport Leslie. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun.ACM*, **21**(7), 558–565.

Lappeteläinen A., Tuupola JM, Palin A., & Eriksson T. 2008. Networked systems, services and information, The ultimate digital convergence. *In: 1st International NoTA conference, Helsinki, Finland.*

Lassila O. 2007 (November). *Programming Semantic Web Applications: A Synthesis of Knowledge Representation and Semi-Structured Data.* Ph.D. thesis, Helsinki University of Technology.

Mattern F. 1989. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 215–226.

Mills D.L. 1995. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networking (TON)*, **3**(3), 245–254.

Oliver I. 2009. Information Spaces As A Basis for Personalising The Semantic Web. *Interational Conference on Enterprise Information Systems (ICEIS'09).*

Oliver I., & Boldyrev S. 2009 (May). *Operations on Spaces of Information.* Tech. rept. Nokia Research Center, Helsinki, Finland.

Oliver I., & Honkola J. 2008. Personal Semantic Web Through A Space Based Computing Environment. *Middleware for Semantic Web 08 at ICSC'08, Santa Clara, CA, USA*, August.

Oliver I., & Honkola J. 2009. A Base-line Semantic Computation Environment for Local Information Spaces. *International Journal of Semantic Computing*, April 7.

Oliver I., Törmä S., & Nuutila E. 2009. Context Gathering in Meetings: Business Processes Meet Agents and The Semantic Web. *The 4th International Workshop on Technologies for Context-Aware Business Process Management (TCoB 2009).*

Postel J. 1981. *RFC 793: Transmission control protocol.* Tech. rept. Defense Advanced Research Projects Agency.

Prechelt L. 2000. An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl. *IEEE Computer*, **33**(10), 23–29.

Rescorla E. 2000. *RFC 2818: HTTP Over TLS.* Tech. rept. The Internet Engineering Task Force (IETF).

Richardson L., & Ruby S. 2007. *Restful web services.* O'Reilly.

Shadbolt N., Hall W., & Berners-Lee T. 2006. The semantic web revisited. *IEEE intelligent systems*, **21**(3), 96–101.

Appendix A

Performance Test Results

	gen_ontology	receive	_poll (1st run)	_poll - receive - gen_ont	receive - gen_ont
Run 1	0,311895	0,795161	1,201739	0,094683	0,483266
Run 2	0,369565	0,920296	1,356698	0,066837	0,550731
Run 3	0,342146	0,778134	1,176526	0,056246	0,435988
Run 4	0,357959	0,788692	1,192466	0,045815	0,430733
Run 5	0,367875	0,815357	1,261488	0,078256	0,447482
Run 6	0,294905	0,867308	1,257144	0,094931	0,572403
Run 7	0,396050	0,873393	1,299802	0,030359	0,477343
Run 8	0,409502	1,035527	1,631453	0,186424	0,626025
Run 9	0,321362	0,782214	1,194077	0,090501	0,460852
Run 10	0,335739	0,765437	1,228414	0,127238	0,429698
Summary					
min	0,294905	0,765437	1,176526	0,030359	0,429698
max	0,409502	1,035527	1,631453	0,186424	0,626025
avg	0,350700	0,842152	1,279981	0,087129	0,491452
stddev	0,029490	0,065583	0,089602	0,031626	0,054961

Table A.1: Performance measurement results for nonexistent SIB ontology, no existing SIB data

	gen_ontology	receive	_poll (1st run)	_poll - receive - gen_ont	receive - gen_ont
Run 1	0,076645	0,442714	0,913044	0,393685	0,366069
Run 2	0,077467	0,452535	0,864829	0,334827	0,375068
Run 3	0,074299	0,390243	0,790818	0,326276	0,315944
Run 4	0,088509	0,570928	0,983291	0,323854	0,482419
Run 5	0,076495	0,502495	0,916916	0,337926	0,426000
Run 6	0,081643	0,459060	0,854966	0,314263	0,377417
Run 7	0,103022	0,477673	0,892701	0,312006	0,374651
Run 8	0,109858	0,554070	0,978595	0,314667	0,444212
Run 9	0,076887	0,611877	1,012287	0,323523	0,534990
Run 10	0,075595	0,626847	1,032169	0,329727	0,551252
Summary					
min	0,074299	0,390243	0,790818	0,312006	0,315944
max	0,109858	0,626847	1,032169	0,393685	0,551252
avg	0,084042	0,508844	0,923962	0,331075	0,424802
stddev	0,009853	0,065669	0,062099	0,014642	0,062972

Table A.2: Performance measurement results for up-to-date SIB ontology, existing SIB data, no ASI data updates