MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

JSC "Kazakh-British Technical University"
Faculty of Information Technologies

"ADMITTED TO DEFENCE"
Chair of CE Department: *Lyazzat B. Atymtayeva*

Writer of dissertation: *Rustem A. Kamun*

MASTER'S DISSERTATION

6M070300 – "Information Systems" specialty

Theme: **"Ensuring faultless web services specifications by developing discrete and modular systems blueprints"**

Scientific supervisor

Timur F. Umarov,
Ph.D. Computer Science,
Associate professor

Almaty, 2013

MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

JSC "Kazakh-British Technical University"
Faculty of Information Technologies

"APPROVED BY"
Chair of CE Department
Lyazzat B. Atymtayeva,
Doctor of physical and mathematical sciences,
Professor

"____" _____ 2013

ASSIGNMENT FOR MASTER'S DISSERTATION

R. Kamun

6M070300 – Information Systems

Theme: "Ensuring faultless web services specifications by developing discrete and modular systems blueprints"

Objectives: Studying the main issues behind the complex communication behaviour in a Web environment, presenting the limitation of the existing solutions, defining the formal theory that challenge these issues and confirming it.

Source data:

- Programming languages: Session-Java, Chor, Java, Python
- Compiler – Polyglot framework

Scientific supervisor

Timur F. Umarov,
Ph.D. Computer Science,
Associate professor

"____" _____ 2013

Almaty, 2013

Abstract

The Master dissertation contains 96 pages, 8 tables, 17 figures, list of sources – 25.

Session-Java (SJ), Web Services Choreography Description Language (WS-CDL), End-Point Projection (EPP), Java Remote Method Invocation (Java RMI), Common Object Request Broker Architecture (CORBA), Web Services Description Language, Web service (WS)

The object of the research of the Master dissertation is to study the main issues behind the complex communication behaviours in a Web environment, introduce the limitation of the existing solutions such as Java RMI, and their XML variants [1], defining the formal theory that challenge these issues and confirming it, making analyses in the context of the presented theory.

The main goal of the dissertation is to be ensured that every concrete process (web service) is correctly functioning and correctly interacting in specified communication behaviour.

To accomplish the defined goal, it was presented the formal theory that describes communication behaviour, using session and session types, in two different ways and analyses their relationship. The first one is the global calculus, an extended form of Choreography Description Language (CDL) [2], a web service description language developed by W3C's WS-CDL Working Group. And the second is the local calculus, originated from the π-calculus [3], one of the representative calculi for communicating processes. This formal theory is an attempt to enhance the quality of software, as it represents "formal blueprints" of how communicating participants should behave and offer a concise view of the message flows.

For checking and evaluation this theory, the Master project attempts to confirm the suitability of the presented theory for business transaction, by introducing real-life business scenarios developed on Session-Java (SJ) [4], an extension to Java implementing Session-Based programming. Each scenario aim to test multiple features of the language and enable us to identify the advantages and disadvantages. The thesis explores the robustness of the language and the scalability as scenarios vary in size but also complexity. In addition we will be targeting

for things such as ease of programming in SJ, clarity of code, any limitations, bugs or non-implementable scenarios.

Аннотация

Магистерская диссертация состоит из 96 страниц, 8 таблиц, 17 иллюстраций, использовано 25 источника.

Session-Java (SJ), Web Services Choreography Description Language (WS-CDL), End-Point Projection (EPP), Java Remote Method Invocation (Java RMI), Common Object Request Broker Architecture (CORBA), Web Services Description Language, Web service (WS)

Объектом исследования магистерской диссертации является изучение открытых проблем взаимодействия бизнес процессов в Web-среде; демонстрация ограниченности и "неуклюжести" существующих технологических решений (Java RMI, CORBA [1]); введение новой формальной теории для решения изученных проблем, а также разработка и качественная и количественная оценка бизнес протоколов для подтверждения введенной теории.

Основная цель диссертации – гарантировать, что каждый конкретный процесс функционирует и взаимодействует согласно описанной спецификации (соглашению).

Для достижения поставленной цели в диссертации представлена теория, описывающая бизнес процессы и их взаимодействие, используя понятия "сессия" и "типизация сессии". Формальная теория описывает поведение бизнес процесса двумя способами:

1. общий анализ основан на теории "Choreography Description Language" [2];

2. частный анализ основан на теории процессов "π-calculus" [3].

Для того чтобы понять, насколько представленная теория пригодна при дизайне спецификации и реализации бизнес процессов, я реализовал бизнес процессы, описанные в Главе 4 магистерской работы на языке "Session-Java" [4]. Цель каждого реализованного бизнес процесса - исследовать надежность и масштабируемость теории. И, наконец, в Главе 5 приведены результаты сравнения "Session-Java" с существующими технологическими решениями.

Contents

List of Figures

List of Tables

1. Introduction

The growing needs for information availability and accessibility present new challenges for application development. Stand-alone applications cannot fulfil the growing needs anymore. There are two forces working in parallel with regard to the need for integration. First, it is necessary to allow for application integration within an enterprise, and second, there are growing needs to ensure inter-enterprise or "business-to-business" integration.

The majority of companies, however, still have existing legacy applications, developed using different architectures and technologies, which have usually not been designed for integration. Companies cannot afford to write off or replace them over night, because they are mission critical; also they cannot afford to develop their entire information systems from scratch in today's business environment.

In addition, companies will undoubtedly need to introduce new applications and systems from time to time. New solutions are usually based on modern architectures, which differ significantly from architectures used by existing legacy applications. These new applications also have to be integrated with existing applications; and existing applications have to be integrated with each other to fulfil the information availability and accessibility goals. To make things even more difficult, there is often a significant investment already in place for a variety of application integration technologies.

It can be clearly seen that integrating applications is a difficult task, may be even one of the most difficult problems facing enterprise application development. To fulfil these integration objectives, several methods, techniques, patterns, and technologies have been developed over the years, ranging from point-to-point integration over enterprise application integration (EAI) and business process management to service oriented architectures (SOA).

The ability to instantly access vital information that may be stored in a variety of different applications may influence the success of a company. For each company, the presence of effective information infrastructure that avoids the need for employees to perform numerous manual tasks like filling in paper forms, and other bureaucracy, is very important. Employees should not have to contend with

such inefficiencies and irritations, such as switching between different applications to get their work done, reentering the same data more than once in different applications, or waiting for data to be processed. Ideally, a well-integrated system should offer **end-to-end support** for business processes with **instant** access to information, no matter which part of the system is used. Similar consideration hold true for companies that want to be successful in e-business or those that want to improve their position in the virtual world.

Companies are realizing the importance of integration at different speeds. Those, who have seen the advantages of integration and understand how to achieve successful integration, are already fully involved in integration projects with many solutions already working. Other companies are aware that integration is important but, although they may have started integration projects, they do not have the results yet, mainly because the integration projects have not been successful. Further still, some companies are only now realizing the importance of integration, and this could in fact be too late for them. Such companies may be looking for ways to achieve integration fast, without spending too much money, and without assigning too many staff members to the integration project. But cutting corners and attempting to implement only the most needed parts of integration in the shortest possible time will most likely result in only partially working solutions at best.

The problem that makes things worse is the fact that *managers are* often *not familiar* with all the complexity hidden behind the integration process. Sometimes, even the "IT people", the architects and developers, do not fully understand the traps behind integration. Most importantly, managers might not understand that integration is a topic that is related to the company as a whole, and not with the IT department only.

Another scenario that leads to the same disorganized approach is when the management of a company does not see the need for integration yet, but the IT department is aware that integration is needed and should be initiated as soon as possible. Therefore, the integration team starts to build a partial solution to solve the most urgent problems. As the integration is not approved from the top, the IT sector does not get enough resources, it does not have enough time and money, and, most significantly, it does not have authorization to start to solve the integration problem globally. Most developers will agree that these are

all-too-familiar situations.

Integration seems to be one of most important strategic priorities, mainly because new innovative business solutions demand integration of different business units, enterprise data, applications, and business systems. Integrated information systems improve the competitive advantage with unified and efficient access to the information. Integrated applications make it much easier to access relevant, coordinated information from a variety of sources. In effect, the total becomes more than the sum of its parts. It's easy to see that integration can be an important and attractive strategic priority.

Typical companies that have existed more than just a few years rarely have integrated information systems. Rather they are faced with a disparate mix of heterogeneous existing systems. Companies will typically have different applications, developed over time. These include:

1. Applications developed inside the company
2. Custom-built but outsourced solutions
3. Commercial and ERP applications

These applications have typically been developed on different platforms, using different technologies and programming languages. Because they have been developed over time, different applications use different programming models. This is manifested through:

1. Combinations of monolithic, client/server, and multi-tier applications
2. Mix of procedural, object-oriented, and component-based solutions
3. Mix of programming languages
4. Different types of database management systems (relational, hierarchical, object) and products
5. Different middleware solutions for communication (message-oriented middleware, object request brokers (ORB), remote procedure calls (RPC), etc.)
6. Multiple information transmission models, including publish/subscribe, request/reply, and conversational
7. Different transaction and security management middleware
8. Different ways of sharing data
9. Possible usage of EDI, XML, and other proprietary formats for data exchange

1.1 Problem Definition and Main Challenges

Another important aspect is the integration that has already been implemented between the existing applications. This includes everything from simple data exchange to use of middleware products. In addition to all this diversity of architectures, technologies, platforms, and programming languages the companies have kept introducing new applications — and with new applications they have introduced modern application architectures too.

As it has been said in introduction, Web Services are rapidly increasing in complexity and range due to their wide applicability to internet users and businesses, and the use of XML (Extensive Markup Language) formalizing the description of data exchange. Recently, an effort has been made to control communication between web services with the introduction of new methods such as standardized Business Transaction Protocols. Session-Based programming uses a typing language to achieve that control and is said to be highly suitable for programming web service communication.

This thesis presents the novel ideas in the form of formal theory of distributed session programming and presents the practical implementation of session-based programming language, shows the main features and compares with already existing solutions. Business transactions in traditional software engineering are understood to be entities with a short life spam operating in a closely coupled context. They are designed for successful completions, without complications such as loss of data arising during exchange. Hence, with minimal control, thousands of transparent transactions can occur within a system every second without troubling the programmer.

The **smooth operation of interaction** between web services is, however, much harder to achieve. The reason behind that is that when we try to extend the concept of traditional transactions to a loosely coupled environment such as the web we find that they're unsuitable. The situation becomes even harder if companies are dealing with applications of a long life spam, running for hours or even days and almost inevitably resulting in a deadlock. Business-to-business interactions often require such long-lived transactions. A need arises for new transaction implementations, more suitable for the web.

Protocols already exist but the recent blossoming of web services, combining

more and more distributed services makes this a very important topic in today's industrial world. In this project we will be focusing on Session-Based Programming, a method of controlling process interactions (represented by sessions). It works by specifying the intended process transaction protocol using *session types* and implementing the interaction using *session operations*. The session implementations will then be verified with the session specifications to guarantee a correct implementation and secure communication between web services. **Session-Java** (SJ) is an extension to Java implementing Session-Based Programming. The language, that has been developed by scientific members from Imperial College (London) and Queen Mary University (London), is still under development but a stable version of it exists and by programming in it I have built several real life scenarios of web businesses interacting. Those scenarios can be used by the SJ designers as programming paradigms or compared with similar implementations in other languages such as for example the Java RMI and CORBA. The thesis is focusing basically on business scenarios, although there is a wide enough subspace to test different aspects of the language: parallel algorithms, real-time communication). This will be the first time that business communication of a sufficient depth will be implemented in SJ, so our scenarios aim to test multiple aspects of the language and enable us to identify its pros and cons. Our priority is that by the end of the project we will have produced complicated programs in SJ. Through these we're aiming to confirm the suitability of Session-Java as an implementation of business to business transactions. We want to explore the robustness of the language and the scalability as scenarios vary in size but also complexity. In addition we will be looking for things such as ease of programming in SJ, any limitations, bugs or non-implementable scenarios.

In fact, SJ is closely tied to the Web Services Choreography Working group [3], currently designing the Web Services Choreography Description Language (WS-CDL) aimed specifically at interactions amongst web services. The plan is that SJ will be used as part of the implementation technologies for the language. Like UML with Java, this could possibly be directly translated to SJ code, and hence generate code skeletons. If, however, there exist any non-implementable scenarios in SJ then they will have to be overcome before that stage is reached.

1.2 Proposed Concept

Use case: *session-based Web services. A client contacts a Web server through HTTP to start an application session. The request is delegated to an application server, and the session continues between the client and the application server using e.g. TCP, SSL.*

Based on the use-case, the Framework must provide a highly extensible platform for transport-independent type-safe object-oriented communications programming on the basis of session abstraction. Its design corresponds to the standard end-to-end principle in network engineering, where the required abstraction, session-based interaction, is realized by communication actions performed at the endpoints, the SJ Runtime instances running over JVMs.

Unlike standard network layerings, sessions are not tied to a fixed network layer, since the same communication abstraction should be maintained over TCP, HTTP, DCCP or even a link-layer protocol. To simultaneously attain extensibility and efficiency in this multi-transport environment, the Framework have to place a thin layer of abstraction on top of each concrete transport. This abstraction, called Abstract Transport, specifies a set of portable low-level communication instructions, to be implemented by each concrete transport. The semantics of language constructs for session programming is realized by interaction services, which are in turn defined over the Abstract Transport, and thus decoupled from individual transports. This decoupling is essential for meeting the extensibility challenges:

1. A new service implemented over Abstract Transport instantly runs over all existing and future transports, without re-implementation for each transport.
2. Symmetrically, a new transport can be seamlessly integrated by implementing the Abstract Transport, instantly available to all existing and future interaction services.

As an example, consider our use-case which work across different transports, such as cross-transport session migration. Implementing such a service over concrete transports would inevitably increase the amount of ground work required for each additional transport, resulting in error-prone, delayed deployment of the new transport.

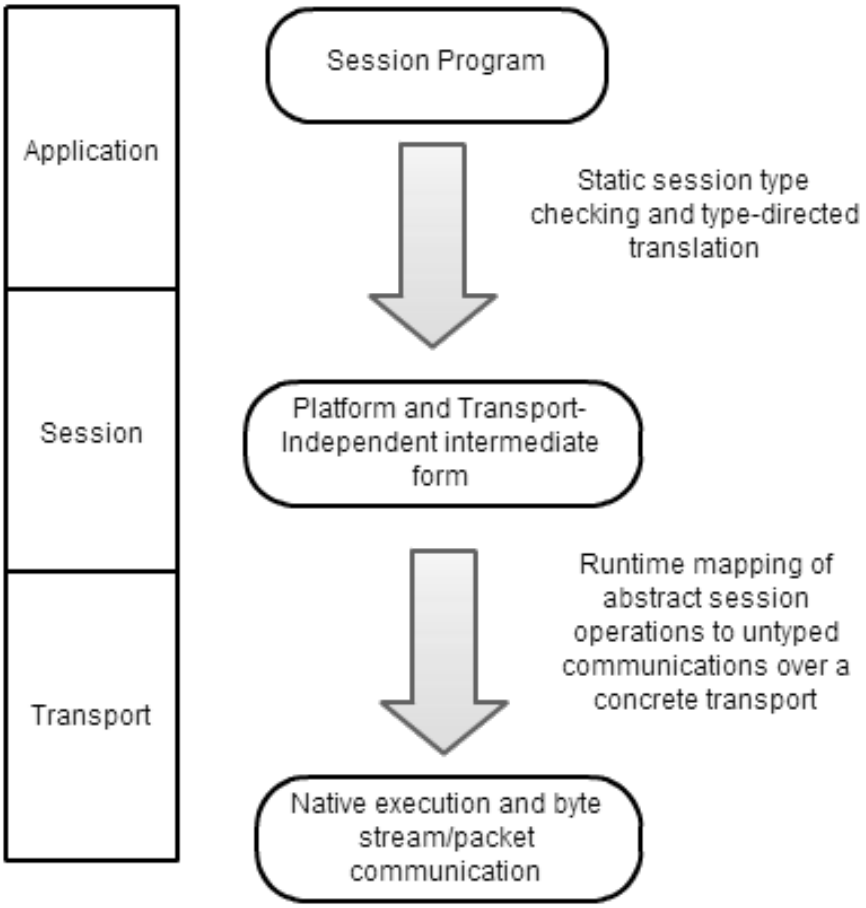

Figure 1.1: Compilation and runtime stages of the Framework

Figure 1.1 depicts the compilation and runtime stages of the Framework, which operate across the layered architecture (session program, interaction services, abstract transport) just discussed. Below there is a brief description of each layer.

Application Layer

The SJ Framework offers the application programmer a rich language facility for transport-independent object-oriented session programming. The compiler, statically type checks programs, and generates a transport-independent intermediate form by translating session operations into calls to the interaction services.

Session Layer

The Runtime has two main responsibilities. The first is performing the interaction services over the Abstract Transport. Services are incorporated into the Runtime as service components. Example: services include session initiation (which validates session peer compatibility), the wire format and serialization for communicating session messages, and cross-transport session migration for delegation as mentioned above.

Abstract Transport Layer

The second role of the Runtime is managing concrete connections, established using available transports, to realize the semantics of the Abstract Transport. The Abstract Transport operations are executed as actions on the underlying transports as directed by transport module implementations.

1.3 Outline of the Thesis

The remainder of this thesis is structured as follows:

Chapter 2: Fundamentals

Chapter 2 gives a background information that helps to understand the design decisions and formal model. The section include short introductions into the following topics: integration middlewares, service-oriented architectures, general standards for Web-service composition, and summarizes the papers that is fundamental to understand the session-based communication.

Chapter ??: Formal Theory

Chapter **??** presents the formal theory of session-based communication structured concurrent programming, that was developed by Nobuko Yoshida, Kohei Honda, Robin Milner and Marco Carbone. The formal theory describes the global interaction as well as end-point behaviours by providing syntax, reduction rules and typing rules.

Chapter 4: Session-based programming and business protocols

Chapter 4 is an implementation part of different business scenarios on e-commerce portal v3na.com. Each scenario consists of interactions and protocol definition, implementation and conclusion parts. The goal of this chapter is to confirm the suitability and applicability of formal theory based on session types.

Chapter 5: Evaluation

Chapter 5 presents the evaluation of the Session-Java by comparing it with Java RMI. Highlights the realization of the most important concepts of session programming.

Chapter 6: Conclusion

Chapter 6 concludes the thesis with a summary and an outlook on future work.

2. Fundamentals

This chapter describes the fundamental basic standards that are used for Web and Enterprise Application Integration today that realized through service-oriented architectures often realized with Web service technologies.

Firstly, we provide some background about integration, integration middlewares and service-oriented architectures. Secondly, we introduce the general standards for service composition: choreography, behavioral interface and orchestration. Since, the thesis ideas based on the concept of Choreography, we will summarize the key papers that discover the choreography essence. Finally, as a prerequisite to formal theory, we present the description of communication behaviour through the Buyer-Seller-Shipper protocol.

2.1 Integration technologies and their evolution

Comprehensive enterprise-wide integration infrastructure usually requires more than one technology. Typically, also, because of the existing technologies, we will have to use a mixture of technologies. When selecting and mixing different technologies, we have to focus on their interoperability.

Interoperability between technologies is crucial because we will use them to implement the integration infrastructure. Achieving interoperability between technologies can be difficult even for technologies based on open standards. Small deviations from standards in products can deny the "on-paper" interoperability. For proprietary solutions, interoperability is even more difficult. Technologies used for integration are often referred to as middleware.

Middleware is system services software that executes between the operating system layer and the application layer and provides services. It connects two or more applications, thus providing connectivity and interoperability to the applications. The middleware concept, however, is today even more important for integration, and all integration projects will have to use one or many different middleware solutions. Middleware is mainly used to denote products that provide *glue* between applications, which is distinct from simple data import and export functions that might be built into the applications themselves.

All forms of middleware are helpful in easing the communication between different software applications. The selection of middleware influences the application architecture, because middleware centralizes the software infrastructure and its deployment. Middleware introduces an abstraction layer in the system architecture and thus reduces the complexity considerably. On the other hand, each middleware product introduces a certain communication overhead into the system, which can influence performance, scalability, throughput, and other efficiency factors. This is important to consider when designing the integration architecture, particularly if our systems are mission critical, and are used by a large number of concurrent clients. There is a large variety of technologies existed today. The most common forms of middleware are:

1. Message-oriented middleware
2. Remote procedure calls
3. Object request brockers
4. Application servers
5. Web services
6. Enterprise service buses

2.1.1 Message-oriented middleware

Message-oriented middleware is a client / server infrastructure that enables and increases interoperability, flexibility, and portability of applications. It enables communication between applications over distributed and heterogeneous platforms. It reduces complexity because it hides the communication details and platform details. Usually the functionality of MOM is accessed via APIs. It provides asynchronous communication and uses message queues to store the messages temporarily. Therefore the interacted applications are loosely coupled. The messages can contain any type of data, asynchronous nature of communication enables the communication to continue even the receiver is temporary not available. The message waits in the queue and is delivered as soon as the receiver is able to accept it. The basic architecture is shown in the Figure 2.1. The well-known MOM technologies are RabbitMQ [5], ZeroMQ [6].

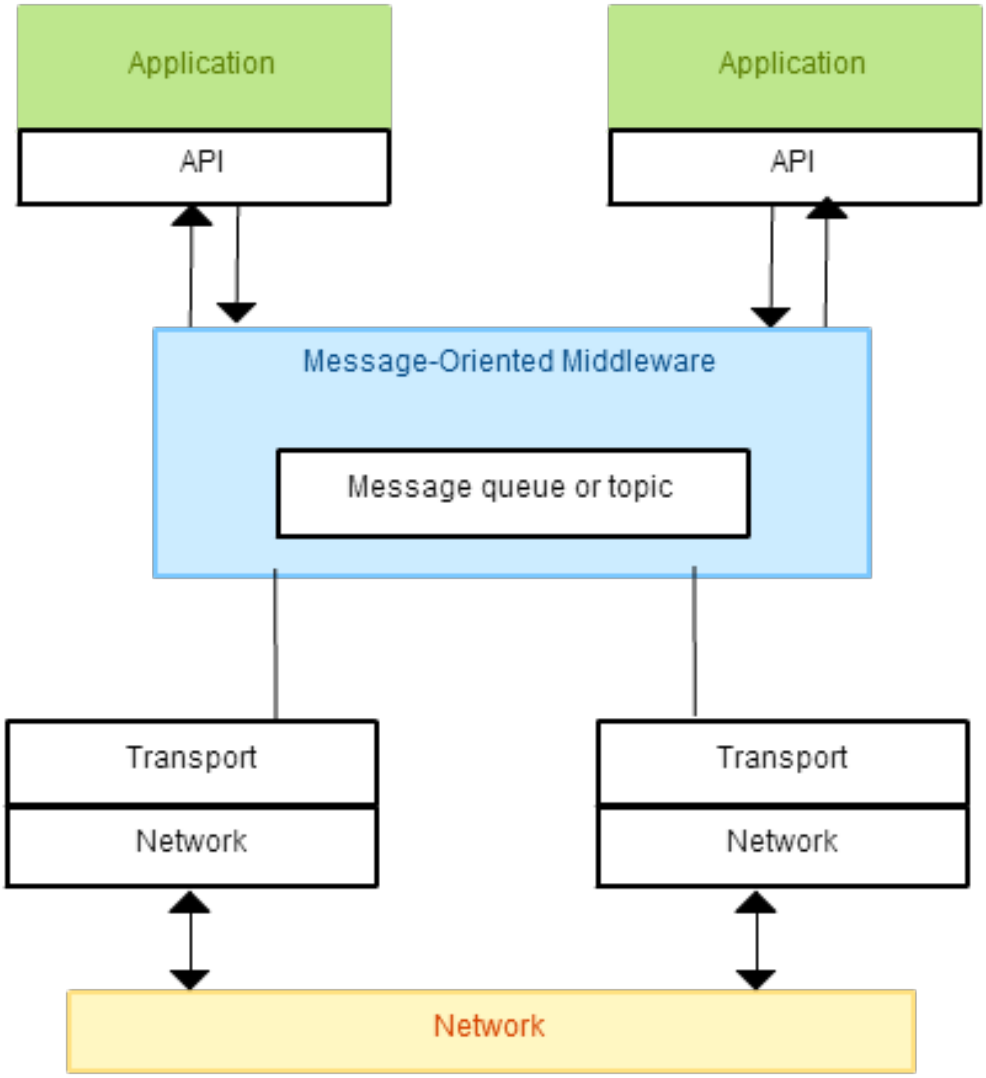

Figure 2.1: Message-oriented middleware

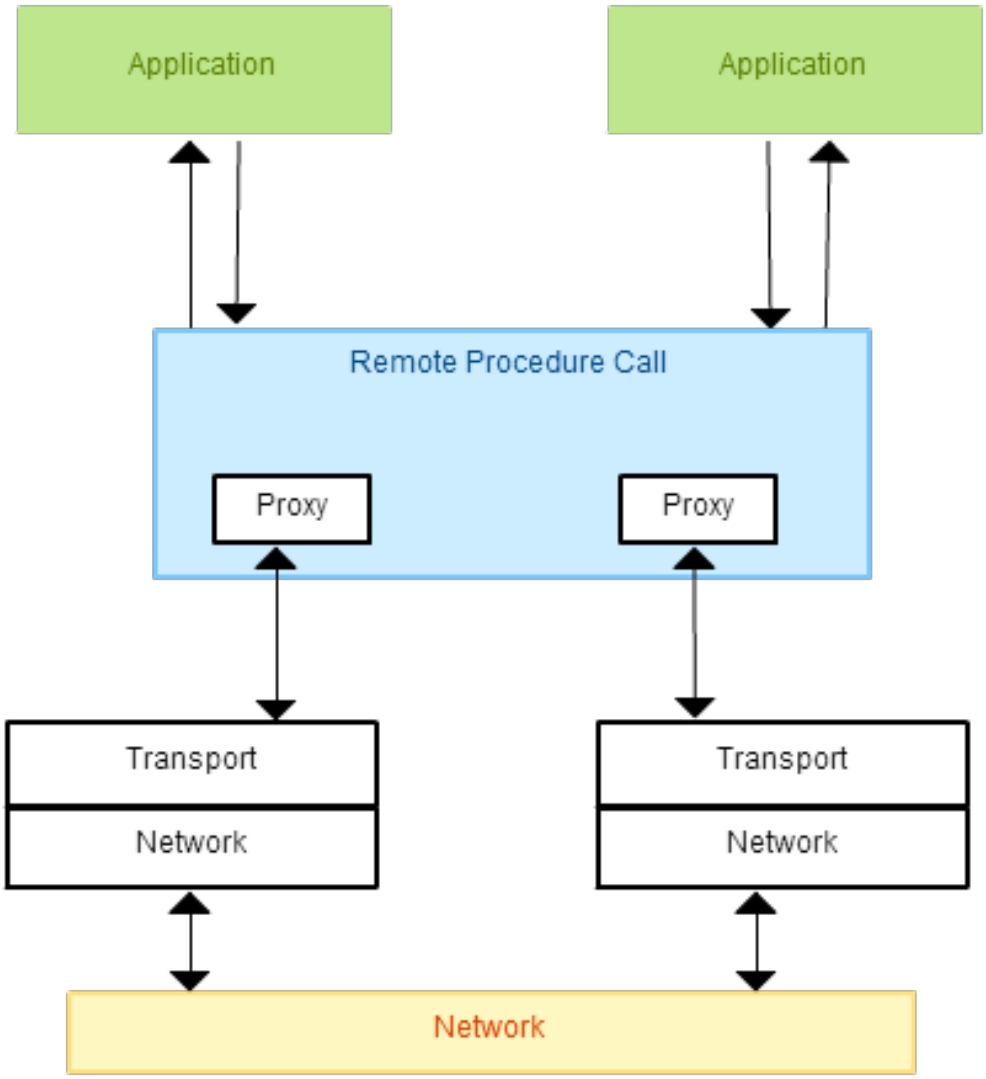

Figure 2.2: Remote Procedure Calls

2.1.2 Remote procedure calls

Remote procedure calls are also a client / server infrastructure intended to enable and increase interoperability of applications over heterogeneous platforms. Similar to MOM, it enables communication between software on different platforms and hides almost all the details of communication. RPC is based on procedural concepts, such as developers use remote procedure or function calls.

RPC increases the flexibility of architecture by allowing a client of an application to employ a function call to access a server on a remote system. RPC allows the remote access without knowledge of the network address or any other lower-level information. The semantics of a remote call is the same whether or not the client and server are collocated. RPC is appropriate for client/server applications in which the client can issue a request and wait for the server to return

a response before continuing with its own processing. On the other hand, RPC requires that the recipient is online to accept the remote call. If the recipient fails, the remote calls will not succeed, because the calls will not be temporarily stored and then forwarded to the recipient when it is available again, as is the case with MOM.

2.1.3 Object request brockers

Object request brokers (ORBs) are a middleware technology that manages and supports the communication between distributed objects or components. ORBs enable seamless interoperability between distributed objects and components without the need to worry about the details of communication. The implementation details of ORB are not visible to the components. ORBs provide location transparency, programming language transparency, protocol transparency, and operating system transparency.

The communication between distributed objects and components is based on interfaces. This enhances maintainability because the implementation details are hidden. The communication is usually synchronous, although is can also be deferred synchronous or asynchronous. ORBs are often connected with location services that enable locating the components in the network. ORBs are complex products but they manage to hide almost all complexity. More specifically, they provide the illusion of locality — they make all the components appear to be local, while in reality they may be deployed anywhere in the network. This simplifies the development considerably but can have negative influence on performance. Figure 2.3 depicts the basic ORB architecture. [1]

ORB products may choose different scenarios as to how and where to implement their functionality. They can move some functions to the client and server components or they can provide them as a separate process or integrate them into to the operating system kernel. There are three major standards of ORBs:

1. ORB CORBA
2. Java RMI
3. Microsoft COM/DCOM

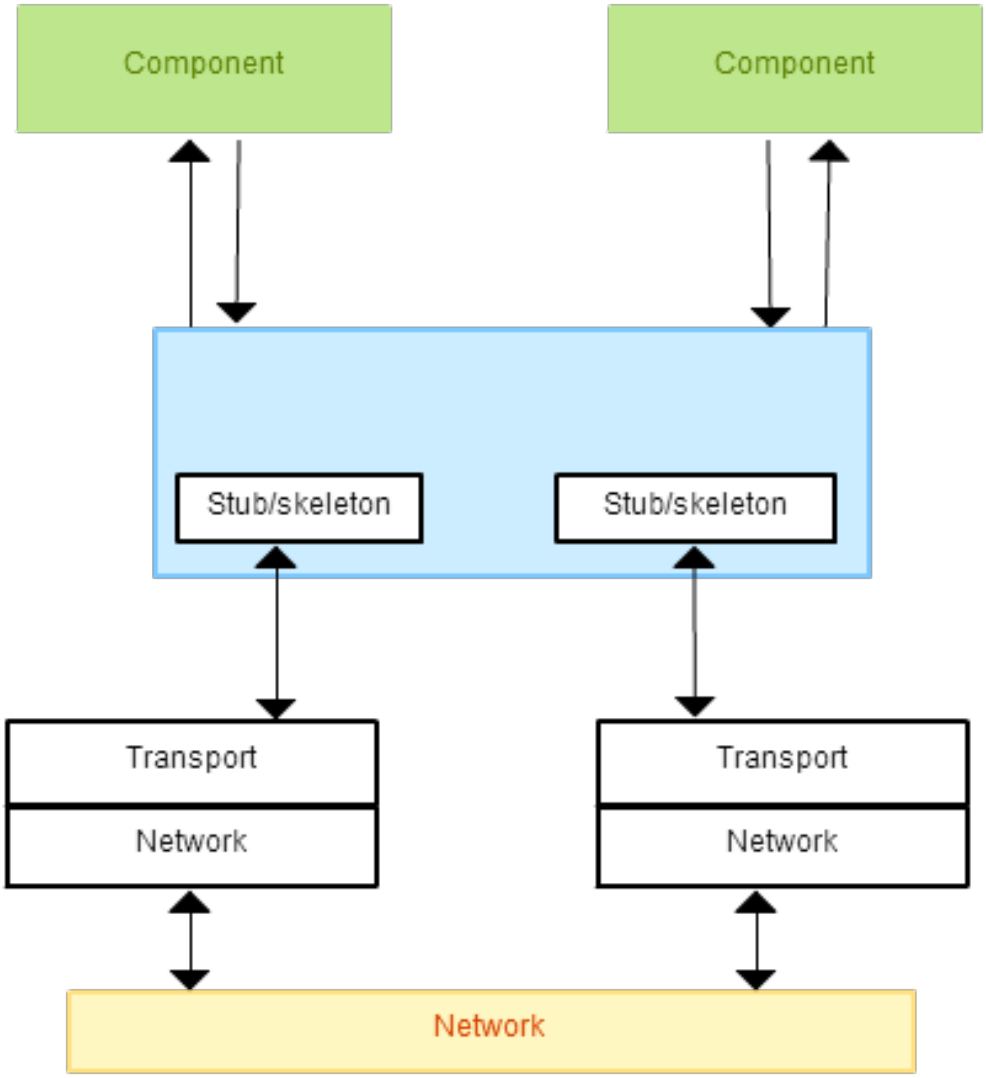

Figure 2.3: Basic ORB architecture

2.1.4 Application servers

Application servers handle the majority of interactions between the client tier and the data persistence tier. They provide a collection of already mentioned middleware services, together with the concept of a management environment in which we deploy business logic components — the container. In the majority of application servers, we can find support for web services, ORBs, MOM, transaction management, security, load balancing, and resource management. Application servers provide a comprehensive solution to enterprise information needs. They are also an excellent platform for integration. Today, vendors often position their application servers as integration engines, or specialize their common purpose application severs by adding additional functionality, like connections to backend and legacy systems and position their products as integration servers. Although such servers can considerably ease the configuration of different middleware products, it is still worth thinking of what is underneath.

The application servers are software platforms, because it is a combination of software technologies necessary to run applications. In this sense, the application servers define the infrastructure of all applications developed and executed on them. Application servers can implement some custom platform, making them the proprietary solution of a specific vendor (these are sometimes referred to as proprietary frameworks). Such application servers are more and more rare.

2.1.5 Web services

Web services are the latest distributed technology that provides the technological foundation for achieving interoperability between applications using heterogeneous software platforms, operating systems, and programming languages. From the technological perspective, web services are the next evolutionary step in distributed architectures. Web services are similar to their predecessors, but also differ from them in several aspects.

Web services are the first distributed technology to be supported by all major software vendors. Therefore, it is the first technology that fulfills the universal interoperability promise between applications running on different platforms. The fundamental specifications that web services are based on are SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), and UDDI

(Universal Description, Discovery, and Integration). SOAP, WSDL, and UDDI are XML based, making web services protocol messages and descriptions human readable.

From the architectural point of view, web services introduce several important changes compared to earlier distributed architectures. They support loose coupling through operations that exchange data only. This differs from component and distributed object models, where behavior can also be exchanged.

Operations in web services are based on the exchange of XML-formatted payloads. They are a collection of input, output, and fault messages. The combination of messages defines the type of operation (one-way, request/response, solicit response, or notification).

Web services provide support for asynchronous as well as synchronous interactions. They introduce the notion of end-points and intermediaries. This allows new approaches to message processing. Web services are stateless and utilize standard Internet protocols such as HTTP (Hyper Text Transfer Protocol), SMTP (Simple Mail Transfer Protocol), FTP (File Transfer Protocol), and MIME (Multipurpose Internet Mail Extensions). So, connectivity through standard Internet connections is less problematic.

In addition to several advantages, web services also have a few disadvantages. One of them is performance, which is not as good as distributed architectures that use binary protocols for communication. The other is that plain web services do not offer infrastructure and quality of service (QoS) features, such as security, transactions, and others, which have been provided by component models for several years.

2.1.6 Enterprise service buses

An Enterprise Service Bus (ESB) is a software infrastructure acting as an intermediary layer of middleware that addresses the extended requirements that usually cannot be fulfilled by web services, such as integration between web services and other middleware technologies and products, higher level of dependency, robustness, and security, management, and control of services and their communication.

An ESB addresses these requirements and adds flexibility to communication between services, and simplifies the integration and reuse of services. An ESB

makes it possible to connect services implemented in different technologies (such as EJBs, messaging systems, CORBA components, and legacy applications) in an easy way. An ESB can act as a mediator between different, often incompatible, protocols and middleware products.

The ESB provides a robust, dependable, secure, and scalable communication infrastructure between services. It also provides control over the communication and control over the use of services. It has message interception capabilities, which allow us to intercept requests to services and responses from services and apply additional processing to them. In this manner, the ESB acts as an intermediary.

An ESB usually provides routing feature to route the messages to different services based on their content, origin, or other attributes and transformation capability to transform messages before they are delivered to services. For XML-formatted messages, such transformations are usually done using XSLT (Extensible Stylesheet Language for Transformations) engine.

An ESB should make services broadly available. This means that it should be easy to find, connect, and use a service independently on the technology it is implemented in. With broad availability of services, an ESB can increase *reuse* and can make the composition of services easier. Finally, an ESB should provide management capabilities, such as message routing, interaction, and transformation.

2.2 Service-oriented architecture – SOA

Services are the main concept of SOA. A service, as it is said, refers to an arbitrary piece of software that offers a well-defined function via standardized interfaces. A service can be reused by other services, legacy applications and customers. This requires that service interfaces and communication protocols must be platform independent. SOA services are loosely-coupled and can be reused whenever their functionality is required. An enterprise service-based architecture include a registry where all available services are listed. The list includes the general contracts of each service as well as technical description to invoke it. As Figure 2.4 depicted, a SOA usually built on three main components: service providers and their services, a registry, and the service users. The service provider pushes the metadata of his services into the central point (registry). Then, user

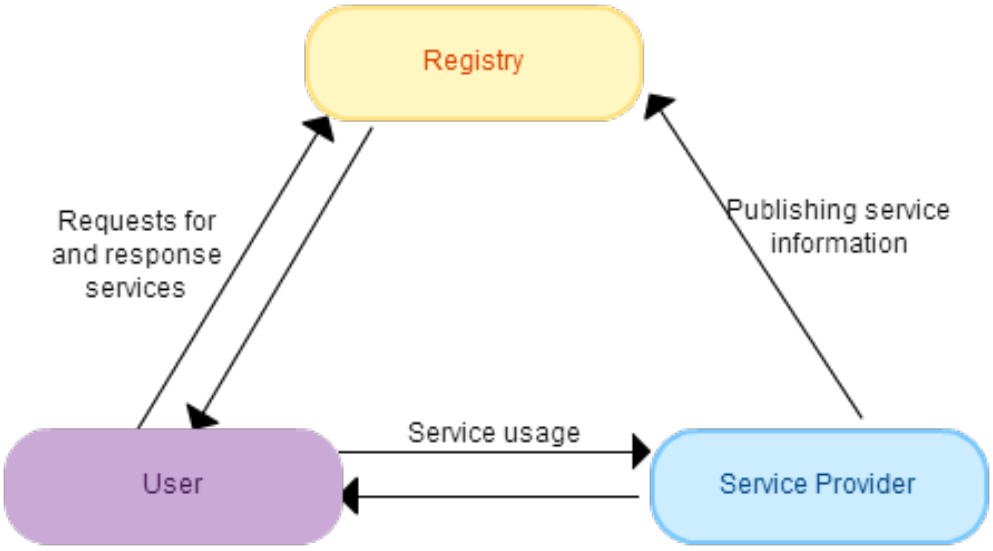

Figure 2.4: SOA components

can search the services buy different criterias (metadata) and, as a result he gets a list of appropriate services and technical specification to invoke them.

Services should be designed according to tasks in business processes and form small and reusable units. Hence, a service possibly needs to access different databases or backend applications. So, this design criteria may be interpreted as a paradigm shift from solution islands, where one application exactly solves one problem.

Generally, nearly all programming languages or technologies for distributed systems can be used, because on a very high level of abstraction only the cross platform communication is a requisite. But in case the SOA should be interoperable, it is necessary that the communication is also independent from the service implementation. For example, Java RMI can be used to create a SOA although integration with other programming languages is difficult. Today, Web services are the de-facto standard used for the realization of SOAs. Such services use SOAP [7] and Internet protocols like HTTP for message exchange. Service descriptions are published using the Web Services Description Language (WSDL) [8]. All these technologies are platform independent and enjoy significant support from the industry. Today, several proprietary and open-source frameworks are available as well as tools for Web service implementation and orchestration of Web services with WSDL and SOAP.

Besides SOAP-based Web services, also other communication protocols are used in modern SOAs, depending on the complexity and requirements. REST

(Representation State Transfer) [9] services have become more and more popular during the last years because the protocol partly works without XML. REST communication is closely related to HTTP protocol, as it supports the following HTTP operations on stateful Web services. A resource is an object that represents the state of the service.

GET: Requests the resource representation. The operation has no effect on the resource state.

POST: Adds and modifies a resource, since this can cause update on the persistence layer.

PUT: Creates new resources or replaces a resource.

DELETE: Deletes a resource.

In contrast to SOAP-based services, REST-ful service does not publish its interface in a standardized way. Instead, setting up a communication requires detailed knowledge about the interfaces. In compliance with the HTTP standard, parameters are submitted via the URL or as HTTP content – normally as XML, JSON. The main advantage of REST services is its simplicity. It guarantees a high scalability because of a small software stack. However, the missing possibility for interface descriptions and the propagation of errors as HTTP error codes hamper a debugging of the communication compared to SOAP.

The loosely coupling of Web services offers various options for service combination as executable workflows. The most used techniques for workflow implementation are **service orchestration** and **service choreography** that we will talk about in the next sections of this chapter. In short, service orchestration is the arrangement of service from a central instance, while service choreography routes messages in a peer-to-peer style directly between the participating services. In industrial scenarious, like the integration of information systems, service orchestration is preferred as a direct mapping of business processes to technical workflow descriptions and monitoring of business processes is possible.

To sum it up, a SOA is a paradigm to integrate different information systems and to integrate legacy systems. All components are implemented as loosely-coupled services. Through orchestration, informal business processes can be

mapped to the technical system level. For example, workflows help to develop a consistent user interface that integrates several backend applications as a single service. The complexity of the involved interfaces in the backend communication is hidden transparently from the user client. Without SOA, the user would have to handle different application clients, each communicating with its own backend counterpart.

2.3 General standards for service composition

As it was mentioned in previous section, there is an increasingly widespread acceptance of SOA as a paradigm for integrating software applications within and across organizational boundaries. Therein, independently designed and operated applications are exposed as Web services. These services are further interconnected using an existing stack of standards including SOAP, WSDL, UDDI, etc. It is important to note, that the main open issue is in managing service interactions that go beyond simple sequences of requests and responses or involve large number of participants. Standardization is a key criteria of SOA. Such standardization initiates as WSDL, SOAP aim to ensuring interoperability between services developed on heterogeneous platforms. Figure 2.5 provides a quick overview of existing WS standards. The standards in the category *composition/processes* deal with the interplay between services and business processes. Today, there are two standardization initiatives: the Business Process Execution Language for Web Services (BPEL4WS or WS-BPEL) [1] and the Web Services Choreography Description Language (WS-CDL) [10]. The significant attention raised by this category of standards discovers the fundamental links that exist between Business Process Management (BPM) and SOA. On the one hand, BPM techniques are used to resource management, process steps description, or capturing the interactions between a process and its environment. On the other hand, a service may serve as an entry point to an underlying business process.

There are three overlapping viewpoints exposed as the standards for service composition:

Behavioural interface (abstract process in BPEL): It captures the behavioral dependencies between interactions in which specified service can engage.

Orchestration (executable process in BPEL): It deals with the description of

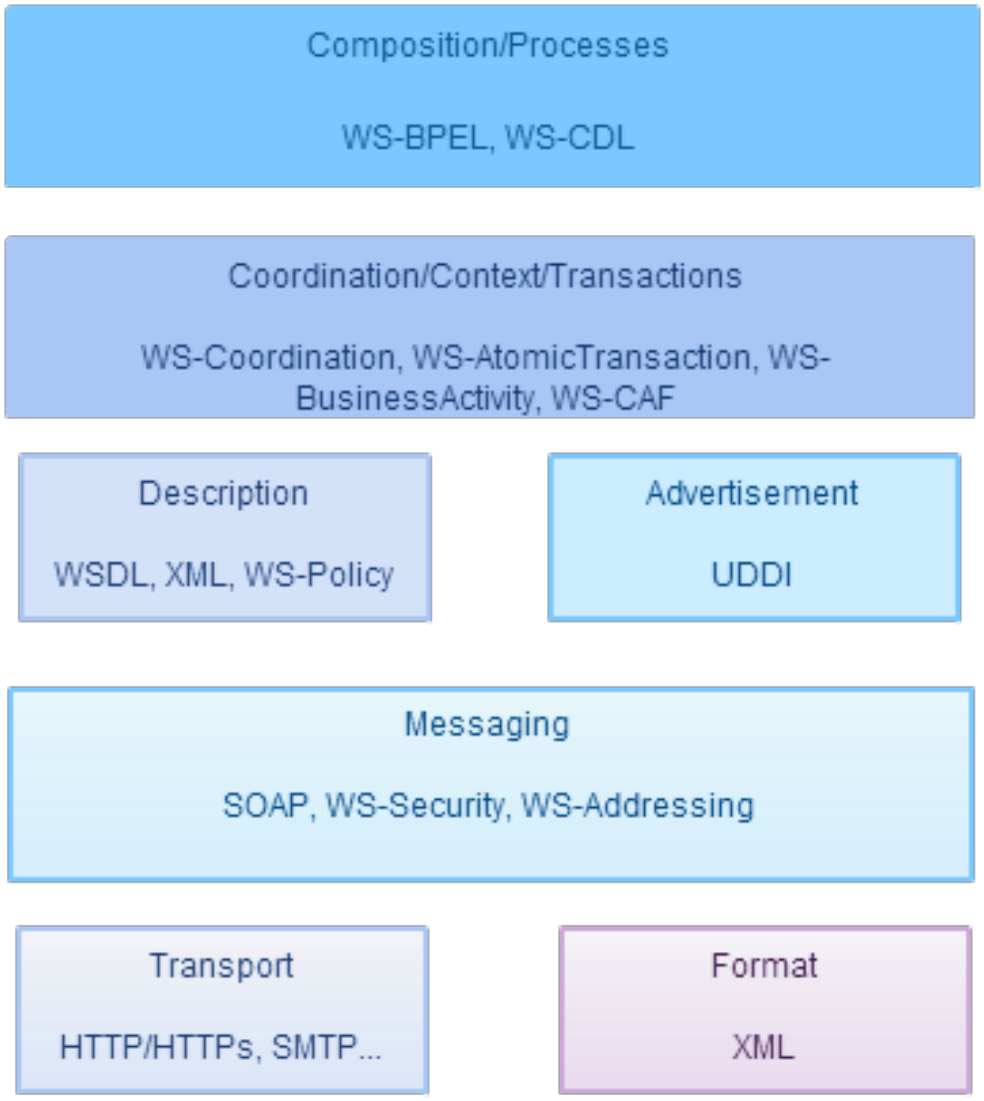

Figure 2.5: WS Standards

the interactions in which a given service can engage with other services, as well as the internal steps between these interactions.

Choreography (multiparty collaboration in ebXML[1]): It describes collaborative processes involving multiple services where the interactions between these services are seen from the global viewpoint.

This thesis will focus on the choreography and behavioral interface viewpoints as a starting point for formal model of global description of interaction and local description of engaged participant. The following subsections provide more precise definitions of the above three viewpoints together.

2.4 Choreography

A *choreography model* describes collaboration between a collection of services in order to achieve a common goal. It captures the interactions in which the participating services engage to achieve this goal and the dependencies between these interactions, including control-flow dependencies (a given interaction must occur before another one), data-flow dependencies, message correlations, time constraints, transactional dependencies, etc. A choreography does not describe any internal action that occurs within a participating service (local viewpoint) that does not directly result in an externally visible effect, such as an internal computation or data transformation. A choreography captures interactions from a global perspective, meaning that all participating services are treated equally. In other words, a choreography covers all the interactions between the participating services that are relevant with respect to its goal.

Figure 2.6 provides an example of a simple choreography through UML sequence diagram. Three services are involved in this choreography: one representing a *buyer*, another one a *seller*, and a third one a *shipper*. The elementary actions in the diagram represent business activities that result in messages being sent or received. For instance, the action *Request For Quote* undertaken by the buyer results in a message being sent to the seller; or depending on the decision of Buyer to accept or reject the quote, seller will continues or terminates the interaction. In addition, every message sending action has a corresponding message receipt action (dual interaction).

[1]Electronic Business using eXtensible Markup Language. Refer to http://goo.gl/qw9hu

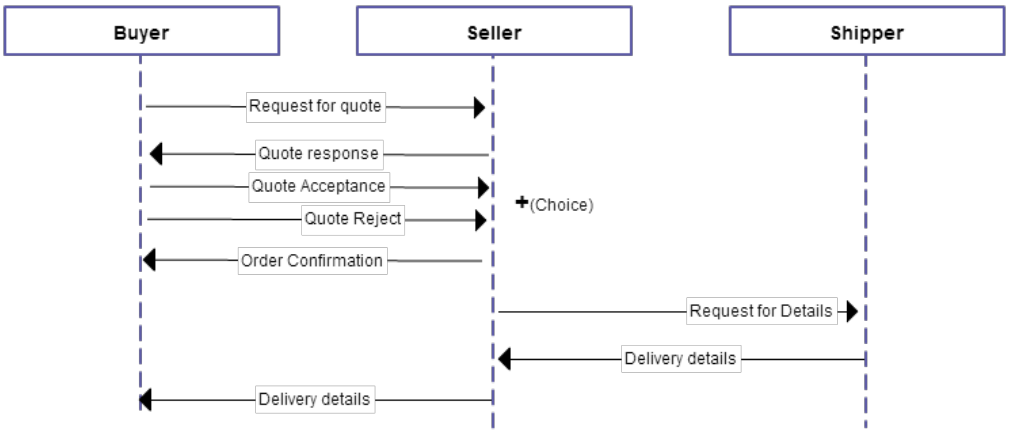

Figure 2.6: Simple BSH protocol

In conclusion, a choreography constitutes an agreement (e.g. interface, contract) between a set of participants as to how a given communication should occur. This agreement may be established by common consultation or designed within a standardization committee.

2.4.1 Behavioural interface

A *Behavioral interface model* captures the behavioral aspects of the interactions in which a particular service can engage to achieve a goal. It complements structural interface descriptions such as those supported by WSDL that capture the elementary interactions in which a service can engage, and the types of messages and the policies under which these messages are exchanged. A behavioral interface captures dependencies between interactions such as control-flow dependencies (e.g., that a given interaction must precede another one), data-flow dependencies, time constraints, message correlations, and transactional dependencies, etc.

Unlike a choreography, a behavioral interface focuses on the perspective of one single party. As a result, a behavioral interface does not capture *complete interactions* since interactions necessarily involve two parties. Instead, a behavioral interface captures interactions from the perspective of one of the participants and can therefore be seen as consisting of communication actions performed by that participant. Like choreographies, behavioral interfaces do not describe internal tasks such as internal data transformations.

Figure 2.7 shows an example of behavioral interface using UML activity diagrams, where activities correspond to message sending and receipt. This be-

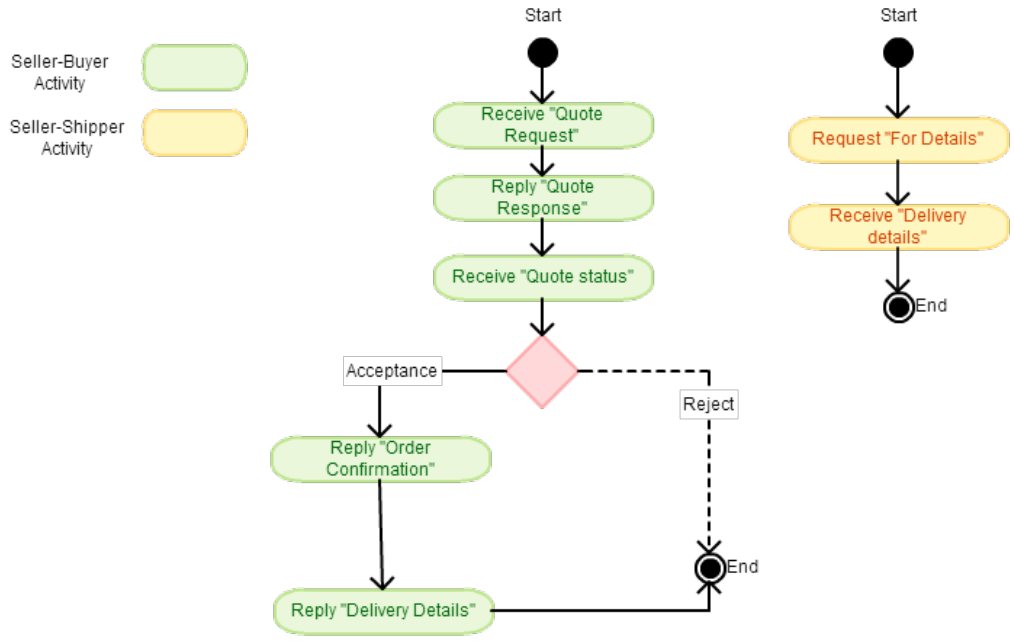

Figure 2.7: Example of behavioural interface

haviour interfaces cover behavior expected from the *seller*'s role in the choreography of Figure 2.6. The left behavioral interface reflects the *seller-buyer* interaction, while the right one reflects the *seller-shipper* interaction.

The example illustrates the fact that in a given Business-to-Business (B2B) collaboration a role in a choreography may be associated with multiple behavioral interfaces. Moreover, given a choreography and a role within this choreography, an arbitrary number of behavioral interfaces may be defined that conform to the behavioral constraints imposed by the choreography on that particular role.

2.4.2 Orchestration

An *orchestration model* describes both the communication actions and the internal actions in which a service engages. Internal actions include data transformations and invocations to internal software modules (e.g., *legacy applications* that are not exposed as services). An orchestration may also contain communication actions or dependencies between communication actions that do not appear in any of the service's behavioral interface(s). This is because behavioral interfaces may be made available to external parties, and, thus, they should only show the information that actually needs to be visible to these parties. Orchestrations are also called *executable processes* since they are intended to be executed by an *orchestration engine*.

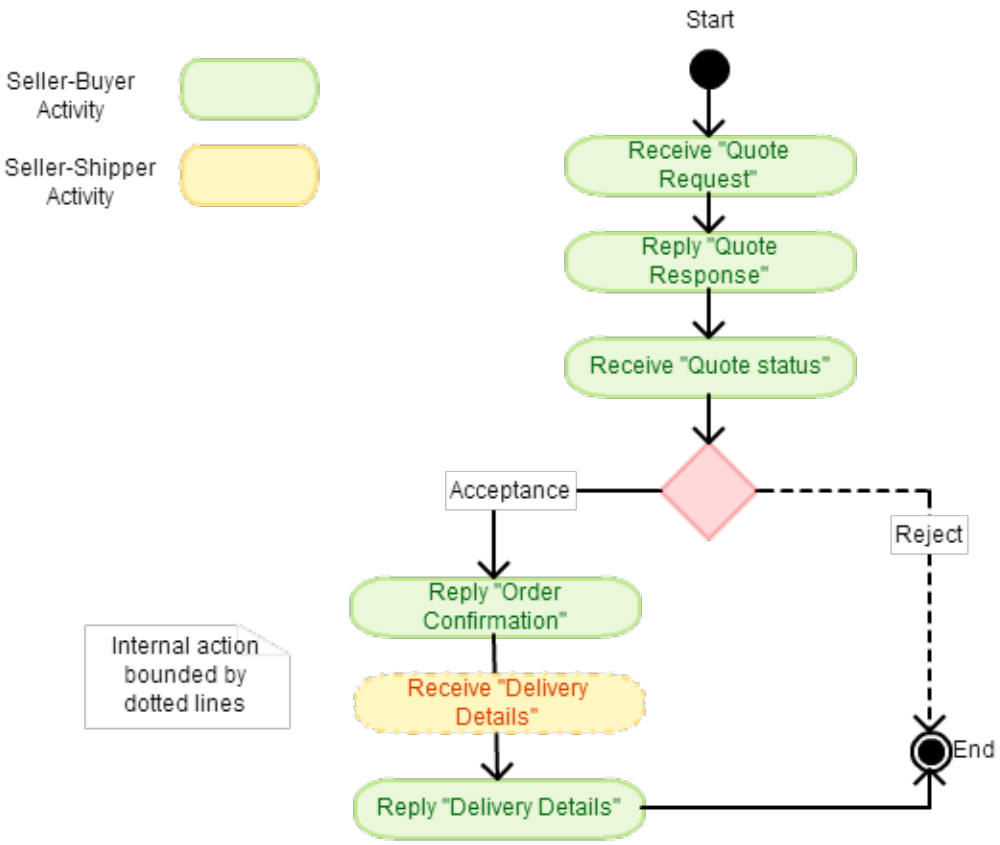

Figure 2.8: Example of an orchestration in the form of an UML activity diagram

Figure 2.8 shows an example of an orchestration in the form of an UML activity diagram. This orchestration adds an *internal action* (shown in dotted lines in the diagram) to the behavioral interface of Figure 2.7.

The viewpoints presented above are overlap that exploited within service composition methodologies to perform consistency checks between viewpoints or to generate code. For example, a choreography model can be used for the following purposes:

1. To generate the behavioral interface that each participating service must provide in order to participate in a collaboration. As explained below, this *behavioral interface* can then be used during the development of the service in question.
2. To check whether the behavioral interface of an existing service conforms to a choreography and, thus, whether the service in question would be able to play a given role in that choreography.

Similarly, a behavioral interface may be used as a starting point to generate orchestration skeleton that can then be filled up with details, regarding internal tasks, and refined into a full orchestration. On the other hand, an existing or-

chestration can be checked for consistency against an existing behavioral interface. For example, it would be possible to detect the case where a given orchestration does not send messages in the order in which these are expected by other services.

2.4.3 WS-CDL: issues and further actions

The paper [11] discovers the issues of the WS-CDL specification. Those issues become the starting point on my continuous research work.

So, one of the requirements of WS-CDL is to provide a means for tools to validate **conformance** to choreography descriptions to ensure **interoperability** between web services. To enable design time or static validation and verification of choreographies to ensure correctness properties such as livelock, deadlock, or to ensure that the runtime behavior of participants conforms to the choreography interface. WS-CDL must be **based on** a **formal language** that provide these validation capabilities.

The existing association between WS-CDL and WSDL is **too restrictive**. A choreography wired to specific WSDL interfaces (either indirectly through references to operations or more directly through an association between roles and their behaviors specified by reference to WSDL interfaces) cannot utilize functionally equivalent services with different WSDL interfaces. In other words, the choreography is statically bound to specific operation names and types, which may hinder the reusability of choreography descriptions. Cast more generally, choreography descriptions which abstractly describe behavior at a higher level, in terms of capability, would allow runtime selection of participants able to fulfill that capability, rather than restricting participation in the choreography to participants based on their implementation of a specific WSDL interface or WSDL operations.

A more subtle dependency is semantic consistency of a global choreography and local process orchestrations. Since a choreography definition introduces message ordering constraints over the interface views of local process orchestration definitions. These need to be supported at the orchestration level in which they are mapped. The expressive power of orchestration semantics, at the same time, should be not be limited by the choreography layer. [11]

Also WS-CDL cannot conveniently support complex multi-party interactions without serializing it.

Finally, WS-CDL is an XML-based language standard. The development of a graphical language for capturing choreographies is not within the current choreography charter of W3C. Any exploitation of WS-CDL should be based on a graphical language, which supports user convenience in capturing specifications, model verification, and validation, as well as configuration for specific deployments utilizing different aspects messaging. Ultimately, it is worth remembering that choreography models are not intended to be directly executed. [11]

2.5 The essence of choreography

Interacting processes must accomplish the goal of the computation. For this, the processes should have correct functionalities and correct interact with each other. With the interaction becoming more complex, the problems related to specify the interaction of the participants will be harder too, if we still want to do it locally. In addition, it is harder to verify the interaction locally. These are the main motivation under the design of WS-CDL. Thus, a deeper understanding of the choreography is very important for the successful development of the web-based computation and application systems. It is better to be done at a suitable abstract level, to make a clear scenery of choreography [12]. The paper [12] defined a small language *Chor*, a model of simplified WS-CDL, and a simple process language for the description of roles from a local viewpoint. Based on these models, the thesis discovers the concept of projections, that map a given choreography C to a set of role processes. By extending a *natural projection* to *restricted natural choreography*, the paper [12] proposed another level of well-formedness.

Using a projection, we will get a set of processes, where each of them represents a role in the choreography. I studied the *local conformance* problem in order to ask on the question: "Does process P can play role R defined by choreography C?".

So, a reasonable definition of the implementation of choreography is based on the concepts of *projection* and *local conformance*. By projection, it should be considered a procedure which takes a choreography in Chor and delivers and delivers a set of processes in the role language, while each of the processes corresponds to a role in choreography. In other words, a projection can partition a

choreography into a set of processes which can mimic the behavior described by the choreography:

$$[[\text{proj}\,(C,\ 1)\,||\ldots||\text{proj}\,(C,\ n)]] = \ [[C]] \tag{2.1}$$

As we will see in the next chapter **??** ("Formal Theory"), these processes make up an implementation of the choreography.

The processes produced by natural projection cannot keep the relative order of their activities, thus, an extra trace of activities appears. So, in order to remedy the ordering problem, it must be inserted extra communication activities in the proper positions to synchronize these processes. The procedure inserts some communication activities in some positions of C, to produce, a revised choreography C', such that we can ensure:

$$[[\text{nproj}(C', 1)||\ldots||\text{nproj}(C', n)]]|\text{acts}(C) = [[C]] \tag{2.2}$$

where $|$ is the filter operation.

2.6 Main criterias for faultless web-services

With a projection defined, it is easy to get a set of processes from a choreography which represent all the roles taking part in the task described by the choreography. In this section, I will turn to define the concept of the implementation of a role, e.g. if a process can play a role defined by choreography, it can take part in the choreography and play with the other valid roles.

The roles of a choreography define a set of requirements on a set of concrete processes (or, web services). In this case, when having a process at hand, we should have a way to decide if this process can play as a specific role. This is called the *conformance* problem. It want to determine if a process conforms with a specific requirement expressed by a role. As said before, what was defined and discussed here can be named *local conformance*, because it want to determine locally a relation between a role (requirement) and a process (a potential implementation).

The following assumptions about a choreography are required to know:

1. It describes all important roles taken part in the task. Each role should be implemented by an distinguishable independent process (an independent

web-service) in the implementation. This also means that there should not be other substantial roles taking part in the interaction, except for the implementation of some local activities for some role(s).

2. It describes all important message passing between the roles. The implementation should not add in other substantial interactions between roles, that is, no additional (substantial) message passing is allowed.
3. The local activities described for each role are important. The implementation of a role (a process or a web-service) must perform the local activities of the specific role in a distinguishable way.

Assume that a choreography C defines random role R, then a process P may be considered as implementation of R, $P \trianglerighteq R$, if

1. P can execute all communication required by R with other roles of C, in suitable time with suitable order.
2. P support all the local activities mentioned in R.

The formal definitions and theorems are also presented in paper [12].

2.7 Why End-Point Projection (EPP) matters

Why does EPP matter? First, with EPP we have a clear idea how a global description can be executed, and, therefore, its computational meaning is clear: a central idea of web services, or in general communication-centred programs and services, is that independently running concurrent agents achieve their application goals through their communication with each other. Thus a global description should be considered as describing behaviour of distributed communicating processes: the latter is the meaning of the former. In this sense, it is only when a uniform notion of EPP is given that the computational content of global descriptions is determined.

Second, EPP offers, for each end-point, what local behaviour a given global description specifies: if we wish to monitor whether an independently developed end-point program behaves in a way specified by a global description, then we can compare the former with the EPP of the latter. Or a programmer/designer working on each endpoint program can check whether it conforms to the original global description with respect to its communication behaviour (such validation, which we already defined as *conformance validation*, will be particularly useful in

collaborative program development).

Thirdly, EPP offer a central underpinning for the theoretical understanding of the structures of global description and their use. Indeed, EPP appears as a link of theories of processes and web service engineering. The established connection enables application of algebras, logics and types of theories of process calculi in the present engineering context. Web service engineering demands theoretical foundations because it is about **interoperability** among organizations with possibly conflicting interests and complex trust relationships that require a clear shared understanding on how they are to interact with each other in a given business protocol. We need a clear criteria as to whether each end-point (organization) is acting conforming to the protocol. It means, that we need clear engineering understanding backed up by a theoretical basis in order to conform the protocol to a regulation.

Chapter 4 provides a general framework for EPP, which can uniformly map a general class of global descriptions onto their end-point counterparts.

The following three are natural formal criteria by which we can measure the effectiveness of an EPP scheme (which are in fact closely related to the two informal criteria we just noted).

1. Important, engineering criteria is that the resulting local descriptions have intuitively a clear and direct connection to the original global description.
2. It is also important to have a general and uniform scheme which can be applied to a large class of global descriptions.
3. Mapping preserves types and other well-formedness conditions.
4. The projected local description implements all behaviours expected from the original global description. Concretely, actions expected from a global description should be faithfully realized by communication among a collection of projected end-points. This property may be called *completeness of EPP*.
5. In the reverse direction, locally projected communicating processes should not exhibit observable behaviour not prescribed in global description, as far as its predefined interface goes.2 Concretely, communications among projected peers should not go beyond actions stipulated in the original global description. This may be called *soundness of EPP*.

3. Formal Theory

The Section 3.1 will outline the key technical ideas using a running example from paper [13]. Sections 3.2 and 3.3 outline the global and end-point calculi. Finally, Section 3.4 describes the theory of end-point-projection.

3.1 Description of communication behaviour

In this section I will show how small, but complex, business protocol taken from [2] can be accurately and concisely described in two small programming languages developed by the authors of the paper [13], one based on global message flows and another based on local, end-point behaviours.

3.1.1 Simple BSH protocol

The starting point is simple business protocol for purchasing a good among a buyer, seller and shipper (Simple BSH Protocol). The expected interaction is described as follows:

1. First, *Buyer* asks *Seller*, through a specified channel, to offer a quote.
2. Then *Seller* replies with a quote. Buyer then answers with either *QuoteAcceptence* or *QuoteRejection*. If the answer is *QuoteAcceptence*, then Seller sends a confirmation to Buyer, and sends a channel of Buyer to Shipper. Then Shipper sends the delivery details to Buyer, and the protocol terminates. If the answer is *QuoteRejection*, then the interaction terminates.

Figure presents a UML sequence diagram of this protocol, but many details are left unspecified.

Main assumptions related to both global and local formalisms:

1. Each *participant* (Buyer, Seller, Shipper) either communicates through channels or change the content of variables local to it.
2. All interactions must be dual, e.g. a sender sends a message and a receiver receives it.
3. Communication can be either an *in-session communication* which belongs to a session, or *session initiation channels* which establishes a session. In

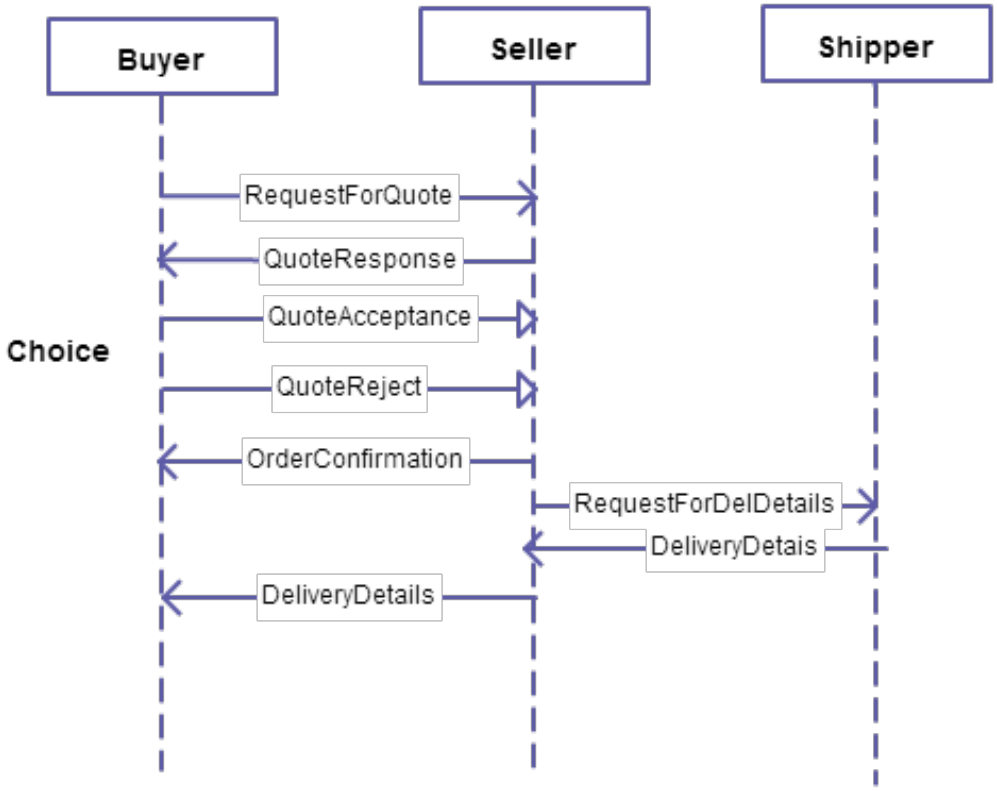

Figure 3.1: Simple BSH protocol

a session initiation communication, one or more fresh session channels are declared (belonging to the same session).

4. A channel can be either a session channel which belongs to a specific session or an session-initiating channel which is used for session-initiation.

Buyer's session-initiating communication in Simple BSH Protocol is described in the global calculus as follows.

$$\text{Buyer} \rightarrow \text{Seller} : \text{quoteCh}(\nu s).I \tag{3.1}$$

which says:

Buyer initiates a session with Seller by interacting through a session-initiation channel quoteCh, declaring a fresh in-session channel s.

The symbol "." indicates sequencing as in CSP [14]. A session initiation can specify more than one session channels, as the following example shows.

$$\text{Buyer} \rightarrow \text{Seller} : \text{quoteCh}(\nu \text{B2S}s, \text{S2B}s).I \tag{3.2}$$

which declares two fresh session channels, one from Buyer to Seller and another in the reverse direction.

In local description, the behaviour is split into two, one for Buyer and another for Seller, using the notations from CSP. For example, equation 3.1 becomes:

$$\text{Buyer}[\text{quoteCh}(s).P1],\ \text{Seller}[\overline{\text{quoteCh}}(s).P2] \tag{3.3}$$

In 3.3, Buyer$[P]$ specifies a buyer's behaviour, while Seller$[P]$ specifies a seller's behaviour. The over-lined channel indicates it is used for output.

An in-session communication specifies an operator and, as needed, a message content. For example quote request and response can be written down as follows:

$$\text{Buyer} \rightarrow \text{Seller:B2S}s(\text{QuoteRequest}).I' \tag{3.4a}$$

$$\text{Seller} \rightarrow \text{Buyer:S2B}s(\text{QuoteResponse}, 3000, x).I' \tag{3.4b}$$

For example, the 3.4b read as follows:

Seller sends a QuoteResponse-message with value 3,000 to Buyer; Buyer upon receipt, assigns the received value, 3,000 to its local variable x.

The description of 3.4a and 3.4b can be translated to end-point behaviours as follows:

$$\overline{\text{B2S}}s(\text{QuoteRequest}).P1,\ \text{B2S}s(\text{QuoteRequest}).P2 \tag{3.5a}$$

$$\overline{\text{S2B}}s(\text{QuoteResponse}).P1,\ \text{S2B}s(\text{QuoteResponse}).P2 \tag{3.5b}$$

In various high-level protocols, there is a situation where a sender invokes one of the options offered by a receiver. A method invocation in object-oriented languages is a simplest such example. In a global communication, it is possible to write an in-session communication that involves such a branching behaviour as follows:

$$\begin{gathered}\{\text{Buyer} \rightarrow \text{Seller} : \text{B2S}s(\text{QuoteAccept}).I1\} \\ + \\ \{\text{Buyer} \rightarrow \text{Seller} : \text{B2S}s(\text{QuoteReject}).I2\}\end{gathered} \tag{3.6}$$

which reads:

"Through an in-session channel B2Ss, Buyer sends one of the two options offered by Seller, QuoteAccept and QuoteReject, and respectively interaction pro-

ceeds to $I1$ and $I2$."

The same interaction can be written down in the local calculus as follows:

$$\text{Buyer's side: } \{\overline{\text{B2S}s}(\text{QuoteAccept}).P1\} \oplus \{\overline{\text{B2S}s}(\text{QuoteReject}).P2\} \quad (3.7a)$$

$$\text{Seller's side: } \{\text{B2S}s(\text{QuoteAccept}).Q1\} + \{\text{B2S}s(\text{QuoteReject}).Q2\} \quad (3.7b)$$

Above $\bigoplus$ indicates that buyer may either behave as $.P1$ or $.P2$, based on its own decision (this is so-called internal sum, whose nondeterminism comes from its internal. Here $+$ indicates this agent may either behave as $.Q1$ or as $.Q2$ depending on what the interacting party communicates through B2Ss (this is so-called external sum, whose nondeterminism comes from the behaviour of an external process).

The global description of Simple BSH protocol is given in Algorithm 1.

Algorithm 1 Global description of Simple BSH protocol

```
1: Buyer → Seller: InitB2S(B2Ss).
2: Buyer → Seller: B2Sch(hQuoteRequest).
3:
4: { Seller → Buyer: B2Sch(hQuoteResponse, vquote, xquote).
5:       Buyer → Seller: (B2Sch hQuoteAccept).
6:       Seller → Buyer: (B2Sch hOrderConfirmation).
7:       Seller → Shipper: InitS2H(S2Hs).
8:       Seller → Shipper: S2Hch(hRequestDeliveryDetails).
9:       Shipper → Seller: S2Hch(hDeliveryDetails, vdetails, xdetails).
10:      Seller → Buyer: (B2Sch hDeliveryStatus, xdetails, ydetails) .0 }
11: +
12: { Buyer → Seller: (B2Sch hQuoteReject) .0 }
```

The local description of Simple BSH protocol is given in Algorithm 2.

3.1.2 Comparison with CDL

In this section, we briefly outline the relationship between CDL and the global/local calculi used in the previous section. The correspondence/difference are summarized in Table 3.1.

Algorithm 2 Local description of Simple BSH protocol

```
1:  Buyer[ InitB2S(B2Sch).
2:       B2Ss(hQuoteRequest).
3:       B2Ss(hQuoteResponce, xquote).
4:            { B2Ss(hQuoteAccept).
5:            B2Ss(hOrderConfirmation).
6:            B2Ss(hDeliveryDetails, ydetails) .0 }
7:            +
8:            { B2Ss(hQuoteReject) .0 }]
9:
10: Seller[ InitB2S(B2Sch).
11:      B2Ss(hQuoteRequest).
12:      B2Ss(hQuoteResponce, vquote).
13:           { B2Ss(hQuoteAccept).
14:           B2Ss(hOrderConfirmation).
15:           InitS2H(S2Hs).
16:                S2Hs(hDeliveryDetails).
17:                S2Hs(hDeliveryDetails, xdetails).
18:           B2Ss(hDeliveryDetails, xdetails) .0 }
19:           +
20:           { B2Ss(hQuoteReject i) .0 }]
21:
22: Shipper[ InitS2H(S2Hs).
23:      S2Hs(hDeliveryDetails).
24:      S2Hs(hDeliveryDetails, vdetails) .0]
```

Table 3.1: Comparison analysis of CDL and global/local calculi

Feature	**CDL**	**FORMALISM**
Session channels	Located at input	No restriction
Session initiation	Implicit	Explicit
General co-relation	Yes	No
Typing	Informal	Formal
Type checking	No	Yes
Local exception	None	Yes
Repetition	Loop	recursion
Sequencing	Imperative	prefix
EPP	Implemented	Proved
Global variable lookup	Yes	No
Global completion	Yes	No
Predicate based	Yes	By adding "when"

3.2 Global message flows

The description of interactions in the global calculus concentrates on a notion of *session*, in which two interacting parties first establish a private connection and do a series of interactions through that private connection, possibly interleaved with other sessions. More concretely, processes first exchange fresh session channels for a newly created session, then use them for interactions belonging to the session (this is equivalent to the more implicit framework where identity tokens in message content are used for signifying a session). This idea has a direct association with a type discipline, where we represent a structured sequence of interactions between two parties as a type. Here type describes an abstract notion of interface of a service, and is inferred by typing rules for each description following its syntactic structure. For example,

$$\text{Buyer} \rightarrow \text{Seller} : s(\text{hRequestQuote}, \text{productName}, xi). \tag{3.8a}$$

$$\text{Seller} \rightarrow \text{Buyer} : s(\text{hReplyQuote}, \text{productPrice}, yi). \tag{3.8b}$$

where, again, a Buyer requests a quote for a product, specifying its name through a session channel s. Then through the same channel s, a Seller replies with the quote value. This interaction at s can be abstracted by the following session type.

$$s \uparrow \text{RequestQuote(String)}.s \downarrow \text{ReplyQuote(Int)} \tag{3.9}$$

The abstraction is given from a buyer viewpoint. Session type for a seller will be in opposite direction. It is important to note, that there is a natural notion of duality associated with session types.

The formal syntax specification is given in [13].

3.2.1 Reduction

Computation in the global calculus is represented by a step-by-step transition, each step consisting of:

1. Execution of a primitive operation, which can be communication, assignment and conditional.
2. Effects the execution above has on the local state of an involved participant.

To formalise this idea, we use a configuration which is a pair of a state (a collection of the local states of all participants involved) and an interaction, written (σ, I). Formally a state, ranged over by $\sigma, \sigma', \ldots$ is a function from $\text{Var} \times P \to \text{Val}$, i.e. a variable at each participant is assigned a value in a store. We shall write $\sigma@A$ to denote the portion of σ local to A, and $\sigma[y@A \to \nu]$ to denote a new state which is identical with σ except that $\sigma'(y, A)$ is equal to ν. The dynamics is then defined in the form:

$$(\sigma, I) \to (\sigma', I') \tag{3.10}$$

which says I in the configuration σ performs one-step computation and becomes I' with new configuration σ'. The relation $\to$ is called reduction relation[1]. The following example demonstrates the reduction relation based on recursion:

$$(\sigma, \text{rec}X^B.x@B := 1.X^B) \to (\sigma[x@B \mapsto 3], \text{rec}X^B.x@B := 1.X^B) \tag{3.11}$$

[1] The term "reduction" originally came from λ-calculus.

All reduction rules are given in [15].

3.2.2 Typing

As briefly mentioned in previous Section, session types [16] are used as the type structures for the global calculus. In advanced web services and business protocols, the structures of interaction in which a service/participant is engaged in may not be restricted to one-way messages or RPC-like request-replies. This is why their type abstraction needs to capture a complex interaction structure of services, leading to the use of session types. The grammar of types presented below.

$$\begin{aligned} &\Theta ::= \text{bool}|\text{int}|\ldots \\ &a ::= \sum_i s \downarrow op_i(\Theta_i).a_i | \sum_i s \uparrow op_i(\Theta_i).a_i \qquad |a_1|a_2|t|\text{rec } t.a|\text{end} \end{aligned} \tag{3.12}$$

Above $q, q', \ldots$ range over value types, which in the present case only includes atomic data types. $a, a', \ldots$ are session types. Note session channels $s, s', \ldots$ occur free in session types (this is necessary because of multiple session channels in a single session, cf. [17]). In order to be commutative and associative with the identity end, there specified '|'. Recursive types are regarded as regular trees in the standard way [18], [19]. The meaning of each construct is given below:

1. $\sum_i s \downarrow op_i(\Theta_i).a_i$ is a branching input type at s, indicating possibilities for receiving any of the operators from $\{op_i\}$ with a value of type Θ_i.
2. $\sum_i s \uparrow op_i(\Theta_i).a_i$, a branching output type at s, is the exact dual of the above.
3. $a_1|a_2$ is a parallel composition of a_1 and a_2, abstracting parallel composition of two sessions. We demand session channels in a_1 and those in a_2 are disjoint.
4. t is a type variable, while rec $t.a$ is a *recursive type*, where rec t binds free occurrences of t in a. A recursive type represents a session with a loop. The main assumption is that each recursion must be guarded, i.e., in rec $t.a$, the type a should be either an input/output type or n-ary parallel composition of input/output types.
5. 'end' is the inaction type, indicating termination of a session. 'end' is often

omitted as we will see in implementation chapter.

Each time a session occurs at a shared service channel, session channels are freshly generated and exchanged. Thus the interface of a service should indicate a vector of session channels to be exchanged, in addition to how they are used. This is represented by abstract session type, or service type, in which concrete instances of session channels in a session type are abstracted, written:

$$(\tilde{s})\alpha \tag{3.13}$$

The next step is to introduce the notion of duality that was mentioned already in this chapter. So, the co-type, or dual, of α written as $\tilde{\alpha}$ is given as follows,

$$\begin{aligned} \overline{\sum_i s \downarrow op_i(\Theta_i).\alpha_i} &= \sum_i s \downarrow op_i(\Theta_i).\overline{\alpha_i} \\ \overline{\text{rec } t.\alpha} &= \text{rec } t.\alpha \\ \bar{t} &= t \\ \overline{\text{end}} &= \text{end} \end{aligned} \tag{3.14}$$

3.2.3 Examples of session types

Example 1. Consider the following interaction, assuming 'adr' and 'prdName' are variables of string type, located at both Buyer and Seller.

1: Buyer → Seller: (s1hQuoteReq, prd, prd).
2: Seller → Buyer: (s2hQuoteRep, 100, y).
3: Buyer → Seller: (s1hPurchase, adr, adr).0

The contract offered by Seller can be described by following session type expression:

$$s \downarrow \text{QuoteReq(string)}.s2 \uparrow \text{QuoteRep(int)}.s1 \downarrow \text{Purchase(string)}.\text{end}$$

While the interface offered by Buyer can be type-abstracted as follows:

$$s \uparrow \text{QuoteReq(string)}.s2 \downarrow \text{QuoteRep(int)}.s1 \uparrow \text{Purchase(string)}.\text{end}$$

Example 2. Assume that Example 1 will be preceded by session initiation. Then let assume that session types for Seller and Buyer are α and $\overline{\alpha}$ from Example 1 correspondingly, at the same time the interaction given in Example 1 is I. Then:

$$\text{Buyer} \rightarrow \text{Seller} : \text{ch}(s1, s2).I$$

Then the service type of Seller at channel sh is given as:

$$(s_1 s_2)\alpha \qquad (s_1 s_2)\overline{\alpha}$$

Example 3. Let Example 1 refine with branching.

$$\begin{array}{c} \text{Buyer} \rightarrow \text{Seller} : s1(\text{QuoteReq, prd, prd}). \\ \text{Seller} \rightarrow \text{Buyer} : s2(\text{QuoteRep}, 100, y). \\ \left(\begin{array}{c} \text{Buyer} \rightarrow \text{Seller} : s1(\text{Purchase, adr, adr}).0 \\ +\text{Buyer} \rightarrow \text{Seller} : s1(\text{NoDanke}).0 \end{array} \right) \end{array}$$

This can be abstracted, from the viewpoint of Seller:

$$\begin{array}{c} s1 \downarrow \text{QuoteReq(string)}.s2 \uparrow \text{QuoteRep(int)}.(s1 \downarrow \text{Purchase(string)}.end + \\ +s1) \downarrow \text{NoThanks()}.\text{end} \end{array}$$

The typing rules are given in paper [15].

3.3 End-point calculus

The end-point calculus, an applied variant of the p-calculus [20], specifies local behaviours of end-points and their composition. For example consider the following term in the global calculus (cf. Example 1):

$$\text{Buyer} \rightarrow \text{Seller} : s(\text{QuoteAccept}, 100, x, ., 0)$$

This global description says that Buyer sends a 'QuoteAccept' message with value 100 to Seller, that Seller receives it, and that Seller saves this value in its local variable x. The end-point calculus describes the same situation as combination of local behaviour, located at each end-point. First there is Buyer's behaviour:

$$\text{Buyer}[\overline{s} \triangleright \text{QuoteAccept}(100).\mathbf{0}]\sigma_B$$

where σ_B is Buyer's local state. Similarly we have Seller's local behaviour:

$$\text{Seller}[s \triangleright \text{QuoteAccept}(x).\mathbf{0}]\sigma_S$$

where σ_S is Seller's local state. Interaction takes place when are concurrently composed, as follows.

$$\text{Seller}[s \triangleright \text{QuoteAccept}(x).\mathbf{0}]\sigma_S | \text{Buyer}[\overline{s} \triangleright \text{QuoteAccept}(100).\mathbf{0}]\sigma_B$$

Let this term be written T. Then the communication event is represented using the following one-step reduction:

$$T \rightarrow \text{Seller}[\mathbf{0}]_{\sigma_s[x \mapsto 10]} | \text{Buyer}[\mathbf{0}]_{\sigma_B}$$

where the seller is updated as a result of communication. Communication in local calculus is organized in the unit of session. To avoid a situation when local behaviours of participants does not correspond to their global specification, the type discipline is used. For example, the expression above can be abstracted as follows:

$$s@\text{Buyer} : s \uparrow \text{QuoteAccept}(\text{int}).\text{end}$$

while that of Seller is abstracted as:

$$s@\text{Seller} : s \downarrow \text{QuoteAccept}(\text{int}).\text{end}$$

Since two signatures are clearly compatible, we conclude the composition is well-typed.

3.3.1 Examples

Simple BSH protocol may have the following end-point version:

$$
\begin{array}{c}
\text{Buyer}[\text{B2S}Chs].s \triangleright \text{RequestForQuote}.s \triangleleft \text{QuoteResponse}(\text{xquote}). \\
s \triangleright (\text{QuoteReject} + \text{QuoteAccept}.s \triangleleft \text{OrderConfirmation}.s \triangleright \text{DeliveryDetails})]a| \\
\text{Seller}[\text{B2S}Ch(s).s \triangleleft \text{RequesForQuote.s} \triangleright \text{QuoteResonse}h\nu\text{quote}i.s \triangleleft \\
(\text{QuoteReject} + \text{QuoteAccept}.s \triangleright \text{OrderConfirmation.S2H}hChhs'i.s' \triangleright \\
\text{RequestDelDetailsBuyer}i.s \triangleright \text{DeliveryDetails}(xDD) \\
s \triangleleft \text{DeliveryDetails}]b| \\
\text{Shipper}[\text{S2S}hCh(s').s' \triangleleft \text{RequestDelDetails}(x\text{Cilent}).s \triangleright \text{DeliveryDetails}(DD)]g)
\end{array}
$$

3.4 Summary

In preceding sections, I have presented example specifications both as a global view in the global calculus and as a local view written in the end-point calculus. In doing so, a global description was always introduced first, and from that the corresponding end-point processes were being recovered. From an engineering viewpoint, these two steps — start from a global description, then extract out of it a local description for each end-point — offer an effective methods for designing and coding communication-centric programs. "It is often simply a pain to design, implement and validate an application that involves complex interactions among processes and which together work correctly, if we are to solely rely on descriptions of local behaviours." This is why such tools as message sequence charts and sequence diagrams have been used as a primary way to design communication behaviour. In fact, the primary concern of the design/requirement of communication behaviour of an application would in general be how global information exchange among processes will take place and how these interactions lead to desired effects: the local behaviour of individual components only matter to realize this global scenario. Thus, in designing and implementing communication-centric software, one may as well start from a global description of expected behaviour, then translate it into local descriptions.

Translating a global description to its end-point counterpart, the process called end-point projection, can however be tricky, because it can be easily produced a global description which does not correspond to any reasonable local counterpart. In other words, if you do not follow good principles, the global description does not in fact describe realizable interaction (not well-formed).

4. Session-based programming and business protocols

Communication is becoming a fundamental element of software development. Web applications increasingly combine numerous distributed services; an off-the-shelf CPU will soon host hundreds of cores per chip; corporate integration builds complex systems that communicate using standardized business protocols; and sensor networks will place a large number of processing units per square meter. A frequent pattern in communication-based programming involves processes interacting via some structured sequence of communications, which as a whole form a natural unit of conversation. In addition to basic message passing, a conversation may involve repeated exchanges or branch into one of multiple paths. Structured conversations of this nature are ubiquitous, arising naturally in server-client programming, parallel algorithms, business protocols, Web services, and application-level network protocols such as SMTP and FTP.

Objects and object-orientation are a powerful abstraction for sequential and shared variable concurrent programming. However, objects do not provide sufficient support for high-level abstraction of distributed communications, even with a variety of communication API supplements. Remote Method Invocation (RMI), for example, cannot directly capture arbitrary conversation structures; interaction is limited to a series of separate send-receive exchanges. More flexible interaction structures can, on the other hand, be expressed through lower-level (TCP) socket programming, but communication safety is lost: raw byte data communicated through sockets is inherently untyped and conversation structure is not explicitly specified. Consequently, programming errors in communication cannot be statically detected with the same level of robustness as standard type checking protects object type integrity.

In the previous chapter, thesis has explored a type theory for structured conversations in the context of process calculi and π-calculus. A session is defined as a conversation instance conducted over, logically speaking, a private channel, isolating it from interference; a session type is a specification of the structure and message types of a conversation as a complete unit. Unlike method call, which

implicitly builds a synchronous, sequential thread of control, communication in distributed applications is often interleaved with other operations and concurrent conversations. Sessions provide a high-level programming abstraction for such communications-based applications, grouping multiple interactions into a logical unit of conversation, and guaranteeing their communication safety through types.

Further in the chapter, we will discover the impact of integrating session types into object-oriented environment like Java. Based on the work [4], [21], we will summarize the central features of the Session-Java (SJ) compilation-runtime framework:

1. Integration of object-oriented and session programming disciplines. The thesis provides a good introduction of concise syntax for session types and structured communication operations in SJ. Session-based distributed programming involves specifying the intended interaction protocols using session types and implementing these protocols using the session operations. The session implementations are then verified against the protocol specifications. This methodology uses session types to describe interfaces for conversation in the way Java interfaces describe interfaces for method-call interaction.
2. Ensuring communication safety for distributed applications. Communication safety is guaranteed through a combination of static and dynamic validations. Static validation ensures that each session implementation conforms to a locally declared protocol specification; runtime validation at session initiation checks the communicating parties implement compatible protocols.

Chapter summary. Section 4.1 provides a fundamentals of SJ syntax and features. In section 4.2 presents different scenarios to demonstrate power and efficiency of the language.

4.1 Introduction to SJ

The purpose: To provide direct language support for (binary) session programming as an extension to Java.

Session programming starts from the description of protocols for interaction (using session types), which can then be concretely implemented using a set of

structured communication operations available on session sockets. The SJ programs guarantees communication safety by:

1. statically verifying that session implementations conform to the protocol specifications;
2. a runtime compatibility validation between peers at session initiation.

Session programming in SJ has the following benefits in comparison to widely used alternatives such as regular socket programming or Java RMI:

1. Byte stream communication through raw network (TCP) sockets and the intended communication protocols have no direct representation in the program, neither as types nor programming constructs. These limitations can make programs more difficult to read and understand, and also to verify.
2. RMI and other RPC-based technologies provide with fixed call-return shape of procedure call, that is not suited to expressing certain interaction patterns.

Session programming is for systems and applications where the parties or components interact according to certain protocols: session types are formal specifications of such protocols. Session types describe structured sequences of interaction including basic message passing, branching, branching and repetition. A session is an instance of a session type, i.e. the unit of interaction encapsulating one run of a protocol. From the perspective of abstraction, each session, is conducted on a separate channel.

Session programming in SJ comprises the following stages:

1. design the protocols by which the parties should interact, and specify them as session types;
2. the protocols are explicitly incorporated into the programs for each party or component: we declare the session types for the interactions to be performed;
3. the actual interaction that comprise a session are implemented using the session programming constructs and operations. The session operations are performed like method calls on session socket objects, endpoint handles in the session channels (sockets).
4. each session implementation is statically verified by the compiler according to the associated protocol specification (type system for session programming);

5. session implementations must respect the property of session linearity (no session operations out of its scope) in terms of control flow and object aliasing.

4.1.1 Protocol Declaration

Session programming begins by declaring the protocol for the intended interaction as follows:

$$\text{protocol name } \{\dots\}$$

where name identifies the protocol, following the standard Java naming rules. Protocols may be declared as both local and field variables, although field protocols are currently restricted to private access. The body of the protocol is session type, given by the syntax rules in Table 4.1, below.

Table 4.1: SJ protocol specification

T	$::= T.T$	Sequencing
	begin	Session initiation
	$! < M >$	Message send
	$?(M)$	Message receive
	$\bigoplus\{L_1 : T_1, \dots, L_n : T_n\}$	Session branching
	$\bigoplus[T]*$	Session iteration.
	rec $L[T]$	Session recursion scope.
	$\#L$	Recursive jump.
	$@p$	Protocol reference.

$$M ::= \text{Object type} \mid \text{Primitive type} \mid T$$
$$L ::= \text{Label}$$
$$\bigoplus ::= ! \mid ?$$

The session type specifies how a session should proceed in terms of the actions that the particular party should perform, i.e. a view of the session from the perspective of that party. The key point is that the implementation of a session is governed by the associated protocol: the SJ compiler statically verifies

session implementations against the protocols. The kind of actions that can be specified include basic message passing, conditional and repeated behaviours. The session type elements that describe these actions are explained below in pairs according to the duality relation between corresponding elements. Session type duality ensures that two parties implement compatible protocols, checked through a runtime validation at session initiation. Duality essentially means when one session party is sending a message, the peer is (or will be) expecting a message of that type (or super type), thus precluding dynamic type errors from message communication.

4.1.2 Interaction sequences and session initiation

If T and T' are session types, then

$$T.T'$$

is the session type that describes first performing the interactions of T followed by the interaction T'.

The action of initiating a new session with a peer is denoted by:

$$\text{begin}$$

Unlike most of the other session type constructors, 'begin' is symmetric, meaning that both parties have the same specification of this action. Note that begin can only appear as the initial prefix at the top level of a session type, since a session can only be initiated once. It also worth noting, there is no explicit session type for the action of closing a session: the end of a session is implicit at the end of a type.

4.1.3 Basic message passing, branching, iteration and recursion

$$!<M> \qquad\qquad ?(M)$$

Sending and receiving a message of a type M are respectively described by the dual types. Message types are serializable Java object types, primitive types and session types. The notion of duality between send and receive includes

session sub-typing, which allows a message subtype to be sent where a super type is expected. In general, the '!' ('?') symbol in session types denotes some form of output (input) action, of which send (receive) is a particular instance.

Session can branch into one of multiple parts: one of the parties is responsible for making the branch decision (which path to select) and the other party must follow. Paths are identified by string labels:

$$!\{L_1 : T_1, \ldots, L_n : T_n\} \qquad ?\{L_1 :!T_1, \ldots, L_n :!T_n\}$$

The type on the left specifies that the session party is decision maker and may select one of the paths labeled from L_1 to L_n. The dual type on the right (where $!T_x$ is dual type of T_x), says the session peer should follow the first party's decision and take the corresponding path. The type below means that whichever branch is selected, the session, will then proceed according to the type T' after the branch has been completed.

$$\bigoplus\{L_1 : T_1, \ldots, L_n : T_n\}.T'$$

$$![T]* \qquad ?[!T]*$$

Sessions may involve the iteration of some subsession. Similarly to branching, one session party is responsible for making the decision at each iteration whether or not to iterate, and the session peer must follow. The decision to iterate or not is made before each iteration, so the sub-session may be performed zero or more times.

Repeated behaviour in session types can also be specified using recursion

$$\text{rec } L[T] \qquad \#L \text{ may occur in } T$$

where L is alphanumeric label and '#' is just a symbol which essentially means "jump" back to the protocol state at which the label was declared (rec L) and repeat T. Recursions with different labels can be nested: can jump back out to any nesting level specifying the appropriate label.

4.1.4 Higher order message types

As mentioned above, message types can themselves be session types. The higher-order communication types additionally express changes in the shape of

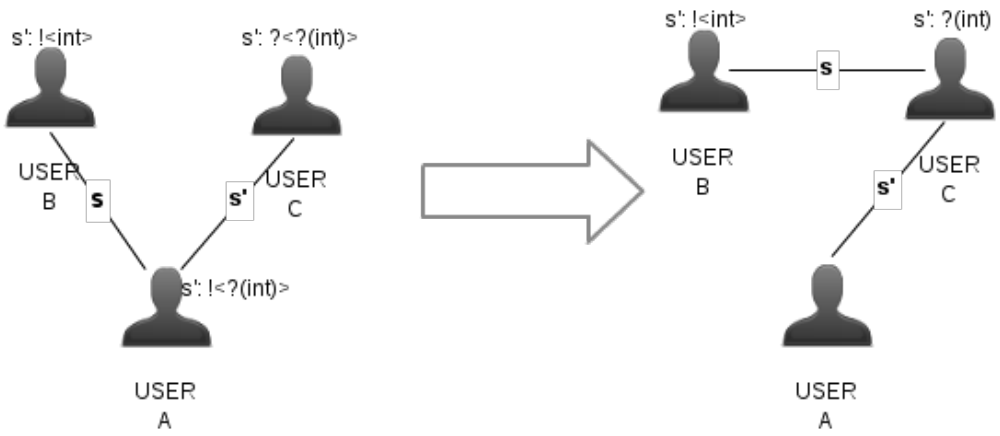

Figure 4.1: Session delegation

the session network. For example, the dual types

$$! <?(\text{int}) > \qquad\qquad ?(?(\text{int}))$$

respectively says that we should send and receive a session of type ?(int). Higher order session communication is often referred to as session delegation. figure 4.1 shows a basic delegation scenario. The left part illustrates the session configuration before the delegation is performed: UserB engaged in a session s of type ! <int> with UserA, while UserA is also involved in a session s' with UserC of type ! <?(int) >. So, instead of accepting the integer from UserB himself, UserA delegates his role in s to UserC, so that he will receive this message. This delegation action corresponds to UserA's higher-order send type for the session s' with UserC. The right part of figure illustrates the change in session configuration after the delegation has been performed: UserB now directly interacting with UserC for the session s.

An important property of delegation is that the type of session being delegated does not itself convey any information about the delegation: the delegation action is between UserA and UserC, and is transparent to UserB, the passive party. Because UserC will fulfill the session contract originally signed by UserA, session delegation keeps communication safety.

Another useful application of higher-order session types is the communication of types like

$$! < \text{begin}.T > \qquad\qquad ? < \text{begin}.T >$$

Since the nested session type is prefixed by begin, the session has not yet been initiated. This communication informs the recipient about another peer from which new sessions of the specified type may be requested. This permits the creation of new edges in a session configuration as well as migration of existing edges. This type of action arises naturally in many real world settings, such as FTP.

For syntactic convenience, one protocol can be referenced from another using the @ operator. For example, given protocol $p1\{\ldots\}$, we can write protocols like protocol $p2\ldots.@p1\ldots$ or protocol $p3\ldots!<@p1>\ldots$ or $p2\ \{\ldots.@p1\ \ldots\}$, where The $@p$ is syntactically substituted for the protocol of that name. Protocols cannot make reference themselves, nor forward references.

4.1.5 Session sockets

Session sockets are implementing the actual session code according to the specified session type (protocol). They are represent the endpoints (participants) of a session connection: each of the parties owns one endpoint and performs the specified interactions via the SJ session operations on that endpoint.

In SJ session sockets are objects that extend the abstract *SJSocket* class. *SJRSocket::SJSocket* and *SJFSocket::SJSocket*, both, employ TCP as underlying transport. SJ is distinguishing session client and server sockets, where the former are used to request sessions from the latter.

Session client sockets are created by calling the static *create* method on the used socket class, *SJFSocket* or *SJRSocket*. This method takes as a parameter an object of type *SJServerAddress*, which binds the IP address and TCP port of the target session server with the type of the session supported by that server. SJServerAddress objects are also created by calling a static create method on the class, passing as parameters the information just described. For example, assuming the protocol p already defined,

```
1 SJServerAddress c = SJServerAddress.create(p, "host", 1234);
2 SJSocket s        = SJFSocket.create(c); // Or use SJRSocket.
```

first creates the session server address c, associated with the protocol (session type) p. c is then used to create a session socket s, which means that s should be used for sessions of type p with specified server by port number and hostname. An

important requirement in SJ is that session-typed objects, such as SJSocket and SJServerAddress, are not permitted to be aliased. In particular, assignment both from and to session-typed variables is forbidden, as is passing the same session-typed variable as multiple arguments in the same method call. Session client sockets cannot be "reused", i.e. once a session has been completed, it cannot be run again using the same session socket object.

Session server-sockets are the counterpart to the above session (client) sockets. Session server-sockets are created in much the same way,

```
1 SJServerSocket ss = SJFServerSocket.create(q, 1234); // 'q' is a protocol.
```

but are created active, meaning that the specified port is opened and *ss* is immediately read to accept session requests.

Sessions are implemented within session-try statements. Session-try is very similar to the regular Java try, but also takes as parameters the session sockets for sessions that may be implemented within its scope. Also, any sessions initiated within a session-try must be completed within the session-try scope.

```
1 try (s1, s2) {
2 ... // someop
3 } catch () {
4 ... // catching exc
5 } finally {
6 ... // session closing
7 }
```

As for regular try statements, the finally clause is optional if there is a catch clause, and vice versa. The implementations of $s1$ and $s2$ may be interleaved along with other Java code, provided that they respect their protocols and the other rules of the session type system. The SJ compiler statically checks that initiated sessions are completed within parent session-try according to the relevant protocols. The completion of the session can be designated by:

1. session delegation;
2. session passing as an argument to another SJ session;
3. spawning in SJ session thread.

4.1.6 Session operations

After creating a protocol (session type) and a session socket that intended to implement that protocol, the session can be implemented within a session-try using the *session operations*, depicted in Table 4.2.

Table 4.2: Session operations specification

s.request()	begin
s.send(m)	$! < M >$
s.receive()	$?(M)$
s.outbranch(L) {P}	$!\{L : T\}$
s.inbranch() {case L1: P1. . . case Ln:Pn}	$?\{L_1 : T_1, \ldots, L_n : T_n\}$
s.outwhile(cond) {P}	$[T]*$
s.inwhile() {P}	$?[T]*$
s.recursion(L) {P}	rec $L[T]$
s.recurse(L)	$\#L$

The session operations are invoked via session sockets in a method call-like manner. To delegate a session, the session socket variable must be passed to a send operation on the target session.

```
1 s1.send(s2) // !<T>, where T is the remaining session type of 's2'
```

Only active session sockets can be delegated, and delegation implicitly completes the delegated session. The receive operation receives delegated sessions:

```
1 SJSocket s2 = s1.receive()
```

Casts are optional, as for ordinary receive operations.

```
1 s2 = (T1) s1.receive()
2 s3 = (@p) s1.receive() // ?(T2), where T2 is the session type declared by p
```

Session sockets can be passed as arguments to, and returned from, session methods. Session methods are the same as regular methods, except the parameter

or return type for session values should the expected session type. The following is an example session method that takes a session type and additional parameters:

```
1 try(s1) {
2     ... // impl of s1
3     finishSession(s1, 1, "hello world");
4 }
5 private void finishSession(T1 session, int cnt, String msg) throws SJIOException
      {
6     ... // finish session according to protocol 'T1'
7 }
```

The key point is that the type of session argument *session*, $T1$, must correspond to the remainder of the session $s1$ at the point of the method call. Session-try scope is served implicitly in session calling method. By the standard rules for exceptions, any exceptions that may be raised by the session operations used, such as SJIOException must be thrown out by the method back to the parent session-try. The next example returns a session from a session method. The session $s2$ is initiated and partially implemented within *getSession* before it is returned to the calling context, where the remainder of the session is implemented.

```
1  try (sret) {
2      ... // 'sret' is null-initialised and not yet assigned to. sret = getSession
          (123);
3      ... // finish 'sret' according to 'T2'.
4  }
5  ...
6  private T2 getSession(int i) throws ... { // Multiple exceptions.
7      ... // 's2' created.
8      try (s2) {
9          ... // 's2' partially implemented.
10         return s2; // 's2' has remaining type 'T2'.
11     }
12     finally { ...
13     }
14 }
```

4.2 Business case studies

V3na.com is an e-commerce web portal that sells SaaS applications for business needs. V3na has developed on Django framework (Python)[1]. The persistence layer is based on MongoDB and Memcached. One of the challenging task was to automate the process of SaaS integration. By integration we understand the following processes with a particular SaaS application:

1. connection: SaaS user can connect SaaS for trial period by simply clicking on the button;
2. subscription extension and freezing;
3. payment confirmation;

Before all the formalisms will be discovered, it is a good starting point to specify simple business protocol of one of the processes just mentioned. For instance, informally the expected process of SaaS connection may be interpreted as follows:

1. SaaS User asks V3na, through a specified channel, to connect particular SaaS application for trial period.
2. V3na service checks that user has an active session with V3na. If so, V3na replies with the messages "SaaS connecting. Wait for couple of minutes". At the same time, V3na builds "User connection" request to this SaaS.
3. SaaS handles the request from V3na: validates message structure, verifies message integrity. After the steps of validations-verifications, SaaS either creates an account with new user for trial period and responds V3na with status "OK" or refuses the request because of failure and responds V3na with status "FAIL".
4. Depending on SaaS response, V3na sends "OK" or "FAIL" message to originally requesting User.

figure 4.2 presents an UML sequence diagram of the connection protocol. Many details are left unspecified: in real interaction, there specified the type of messages exchanged, security protocol, message structure and so on. Further, the communication can be represented through the sessions. In order to represent the communication through the sessions, the following assumptions have to be considered according to [13]:

[1] Django is a high-level python web framework that encourages rapid development and clean. Further information: https://www.djangoproject.com/

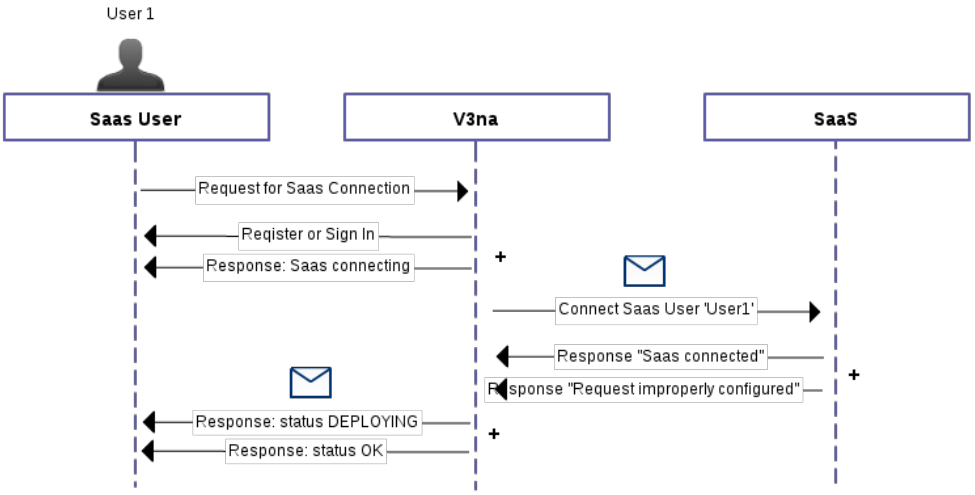

Figure 4.2: V3na protocol for SaaS connection

1. SaaS User, V3na and SaaS are called *participants*;
2. Each participant communicates through the channels
3. Dual communication — a sender sends a message and receiver receives it
4. Communication can be in-session communication, which belongs to a session, or session initiation channels which establishes a session

In the next subsections, I will present the three case studies, increasing in complexity and reflecting the central aspects of session-based programming.

4.2.1 Scenario # 1

This scenario corresponds to the description of connecting user to SaaS on v3na.com in previous section. The scenario is as follows:

1. User begins a request session (s) with cloud service (V3na) and sends the request "Connect SaaS" as JSON-encoded message.

2. V3na sends either:

3. FAIL, if user has no active session (not signed in on V3na) and further interaction terminates

4. OK, if user has logged in and request data has passed validation steps. Then Cloud initiates a new session (s') with SaaS and requests it for new user connection with HttpRequestJSONMessage.

5. If OK label take place, Cloud initiates a new session (s') with SaaS and requests it for new user connection with HttpRequestJSONMessage. finally SaaS responds to Cloud with connection status (OK, FAIL) and V3na sends this status to User. Both sessions have to be terminated.

Protocols. The decision in the protocol will be incorporated through the use of outbranch. So the whole scenario is presented on Table 4.3.

Table 4.3: Protocols of scenario # 1

User	**Cloud (V3na)**	**SaaS**

`protocol p_uv {`
`  begin.`
`  !<JSONMessage>.`
`  ?{`
`  OK: ?(`
`  JSONMessage)`
`  .?(int),`
`  FAIL:`
`  }`
`}`
	`p_vu {`
	`  begin.?(`
	`  JSONMessage)`
	`  .!{`
	`  OK: !<`
	`  JSONMessage`
	`  >.!<int>,`
	`  FAIL:`
	`  }`
	`protocol`
	`  http_req_rep {`
	`  !<JSONMessage>.`
	`  ? (JSONMessage)`
	`}`
	`protocol p_vs {`
	`  begin.`
	`  @http_req_rep`
	`}`
		`protocol p_sv {`
		`  begin.`
		`  ?(JSONMessage). !<`
		`  JSONMessage>`
		`}`

Interactions. The general syntax for global description has been interpreted into a sequence UML diagram, as depicted in figure 4.3. The whole syntax is on the down-left side of the figure. In case of choice, terminated branches are out of scope of the main picture, but still a subpart of the whole diagram. Next step is implementation of this diagram in Session-Java.

The main concept of SJ is that User choice reflected through the outbranch case. For the full source code, refer to Appendix A or repository [22].

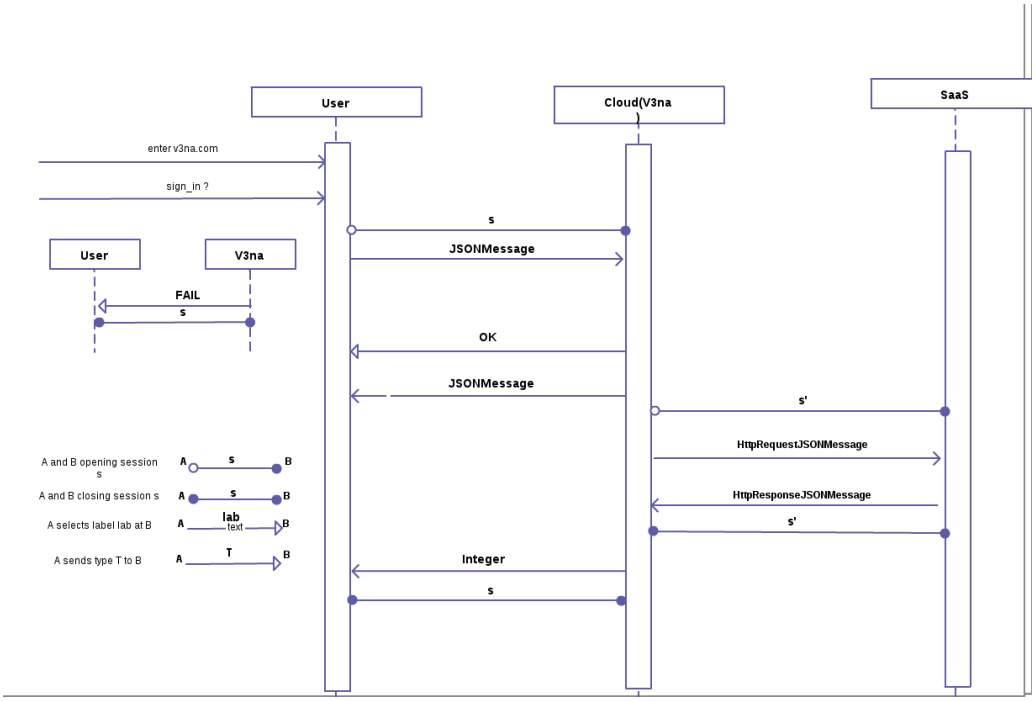

Figure 4.3: Overview of interactions for Scenario # 1

4.2.2 Scenario # 2

New scenario is a bit harder in complexity, and a loop and session delegation features are introduced. The description of this scenario presented below:

1. User begins a request session (s) with cloud service (V3na)
2. V3na asks User to login, so next User provides V3na with login and password Strings.
3. V3na receives User credentials and verifies them: If User is authenticated with minimal amount of tries or amount of tries is out of limit, he is allowed to continue further interactions with V3na, otherwise — not. Go back to step 2.
4. If User is not allowed to access V3na, the interaction between User and V3na continues on DENY-branch, otherwise — on ACCESS-branch.
5. If next branch is ACCESS, User sends his connection request with details to V3na. V3na creates new session with SaaS (s') and delegates the remaining session s with User on the latter and sends last user request details. Session s' is terminated.
6. SaaS continues interaction with user by session s. By steps of validation-verification, SaaS either responds User to proceed interaction by branch OK or FAIL. In both cases User receives from SaaS directly the reason and

status of his request. Session s is terminated.

Protocols. first of all, the protocol provided with iterations using ![...]* ?[...]*. Then protocol introduces higher order operations of type $! < T > \ ? < T >$. Full description is provided in tables 4.6 and 4.5.

Table 4.4: User-Cloud protocols

User	**Cloud**

`protocol p_uv {`
`  begin.?[!<String>.!<String>`
`  ]*.`
`  ?{`
`  ACCESS: !<JSONMessage>.`
`  ?{`
`  OK: ?(JSONMessage), FAIL:`
`  ?(JSONMessage)`
`  },`
`  DENY: ?(String)`
`  }`
`}`
	`private protocol p_vu {`
	`  begin.`
	`  ![ ?(String).?(String) //`
	`  login password`
	`  ]*.`
	`  !{`
	`  ACCESS: ?(JSONMessage).`
	`  !{`
	`  OK: !<JSONMessage>, FAIL:`
	`  !<JSONMessage>`
	`  },`
	`  DENY: !<String>`
	`  }`
	`}`

Table 4.5: Cloud-SaaS protocols

Cloud	**Saas**

protocol p_vs {
  begin.
  !< !{
  OK: !<JSONMessage>,
  FAIL: !<JSONMessage>
  } >.!<JSONMessage>
}
	protocol p_msg {
	  !{
	  OK: !<JSONMessage>,
	  FAIL: !<JSONMessage>
	  }
	}
	
	protocol p_sv {
	  begin.?(@p_msg).?(JSONMessage)
	}

Unlike the previous protocol, the Cloud-Saas protocol significantly altered, also authentication process is added to the protocol in interaction between User — Cloud. It is important to note that

```
1 !<
2     !{
3         OK: !<JSONMessage>,
4         FAIL: !<JSONMessage>
5     }
6 >
```

corresponds to a higher-order message. The $!< \cdots >$ means that it is the Cloud that is passing the high order message and everything inside it is the protocol of the session that Saas should perform with the User. In Saas — Cloud, the protocol defined in more subtle way containing higher order messages by first defining them and then including them in the protocol. For syntactic convenience, one protocol can be referenced from another using @ operator. The $@p$ is syntactically substituted for the protocol of that name.

Interactions. figure 4.4 shows the general picture of session delegation for scenario 2. The left picture is before the session delegation, while the right picture is the state after delegation. figure 4.5 depicts the protocols provided above using an UML sequence diagram. The language of the artifacts has already presented

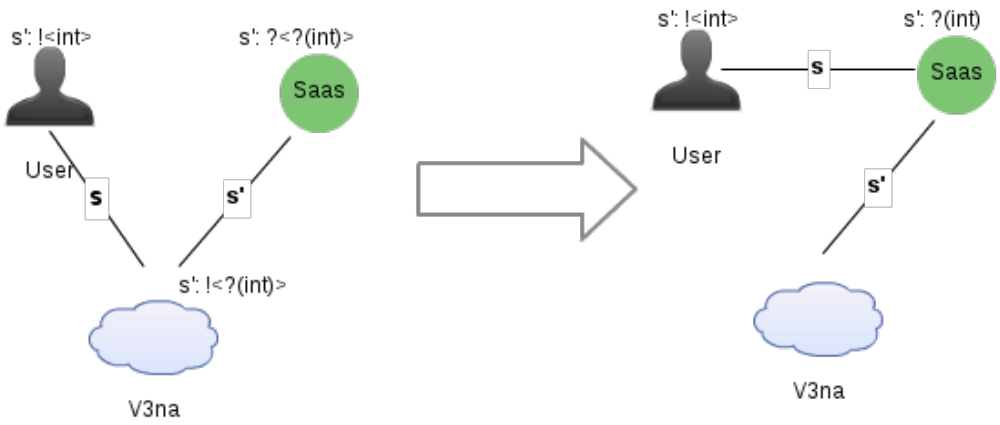

Figure 4.4: Session delegation in Scenario # 2

in the first scenario.

Implementation. Despite the fact that session delegation takes place, the program still remains very simple. Actually delegating the protocol is straightforward and only consists of passing the socket to service:

```
1 s_vs.send(user_vu); // pass the remaining protocol
```

I have decided to include the whole segment of code to illustrate the following point.

```
1  user_vu.outbranch(ACCESS) {
2    JSONMessage req_info = user_vu.receive(); SJServerAddress addr_vs =
         SJServerAddress.create(p_vs, saas_hname, saas_port);
3    SJSocket s_vs = SJRSocket.create(addr_vs); try(s_vs) {
4      s_vs.request();
5      s_vs.send(user_vu); // pass the remaining protocol
6      s_vs.send(req_info);
7    } catch(UnknownHostException uhe) {
8      uhe.printStackTrace();
9    }
10 }
```

To receive a high order message type casting must take place in the case of a protocol, the type of protocol must be explicitly defined:

```
1 v3na_user_socket = (@p_msg) v3na_sv.receive();
```

Where p_msg is defined in the protocols section. This is a reason why it is good practice to first exclusively write the protocol to be delegated and then

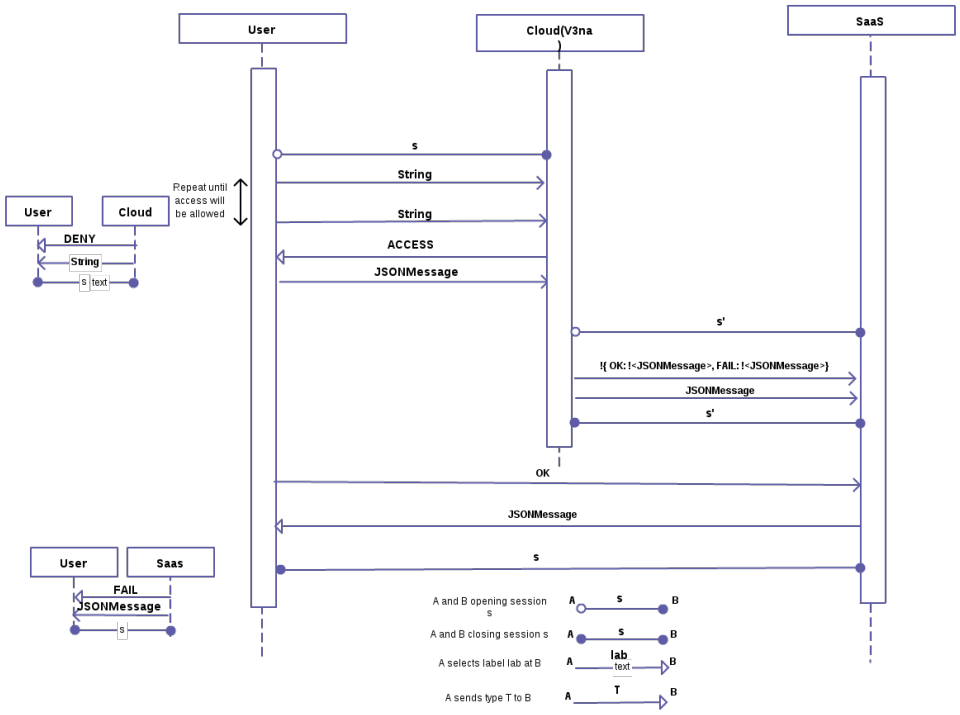

Figure 4.5: Sequence diagram of interactions for Scenario # 2

include it in the final protocol. For the full source code, refer to Appendix B or repository [22].

4.2.3 Scenario # 3

This is a new scenario, devoted to payment and wallet recharging processes analysis developed for V3na.com [23]. The description of this scenario is defined below as follows:

1. User begins a request session (s) with cloud service (V3na)

2. V3na asks User to login, so next User provides V3na with login and password strings.

3. V3na receives User credentials and verifies them: If User is authenticated with minimal amount of tries or amount of tries is already out of limit. In first case, he will be allowed to continue further interactions with V3na, in second case – he will not. Go back to previous step.

4. If User is not allowed to access V3na, the interaction between User and V3na continues on DENY-branch. User receives the error message and protocol is terminated.

5. If next branch is ACCESS, User proceeds with payment or wallet recharging interactions.

6. If user chooses PAYMENT:

   (a) Cloud delegates the remaining interaction of session s to the payment backend via session s'.

   (b) As response from the payment backend, User receives the Goods collection that have to be payed. User chooses either Visa/Mastercard or Non-cash payment way. In both cases User sends his private details: card or transfer details.

   (c) finally User receives the status of his payment and session s terminated

7. If user chooses WALLET:

   (a) Cloud delegates the remaining interaction of session s to the wallet backend via session s'.

   (b) Wallet backend continues interaction with User. User sends taxation number and then his wallet id.

   (c) Wallet backend verifies by taxation number that payment is active and checks by wallet id that user exists.

   (d) finally Wallet backend notifies user with status of wallet recharging and session s is terminated.

Protocols. Calculating the protocols for this scenario is a little different than what has been shown before. In the current scenario the session between Cloud and Cloud backends (Payment, Wallet) involves the Cloud delegating the session with the User to one of those backends. It is similar to an application server behavior, while dealing with multiple clients it must delegate particular work on the its components. The protocol definition includes four protocols (participant per protocol). The protocols are provided in tables 4.6 and 4.7.

Table 4.6: User, Cloud protocols

User	Cloud

protocol payment { ?(Goods).!{
VISA_MASTER: !<CardDetails>,
TRANSFER: !<TransferDetails>
}.?{
PAID: ?(String), DECLINED:
?(String), FAILED: ?(
String)
}
}
private static protocol wallet {
?(String).!<Integer>.!<Integer
>.?{
PAYMENT_INACTIVE: ?(
OSMPMessage),
USER_NOT_FOUND: ?(
OSMPMessage),
OK: ?(OSMPMessage)
}
}
private static protocol p_uv {
begin.?[!<String>.!<String>
]*.?{
ACCESS: !{
PAYMENT: @payment,
WALLET: @wallet
},
DENY: ?(String)
}
}
	protocol p_payment {
	!<Goods>.?{
	VISA_MASTER: ?(CardDetails),
	TRANSFER: ?(TransferDetails)
	}.!{
	PAID: !<String>, DECLINED:
	!<String>, FAILED: !<
	String>
	}
	}
	protocol p_wallet {
	!<String>.?(Integer).?(Integer
	).!{
	PAYMENT_INACTIVE: !<
	OSMPMessage>,
	USER_NOT_FOUND:
	!<OSMPMessage>, OK: !<
	OSMPMessage>
	}
	}
	protocol p_vp {
	begin.!<String>.!< @p_payment
	>
	}
	protocol p_vw {
	begin.!<String>.!< @p_wallet >
	}

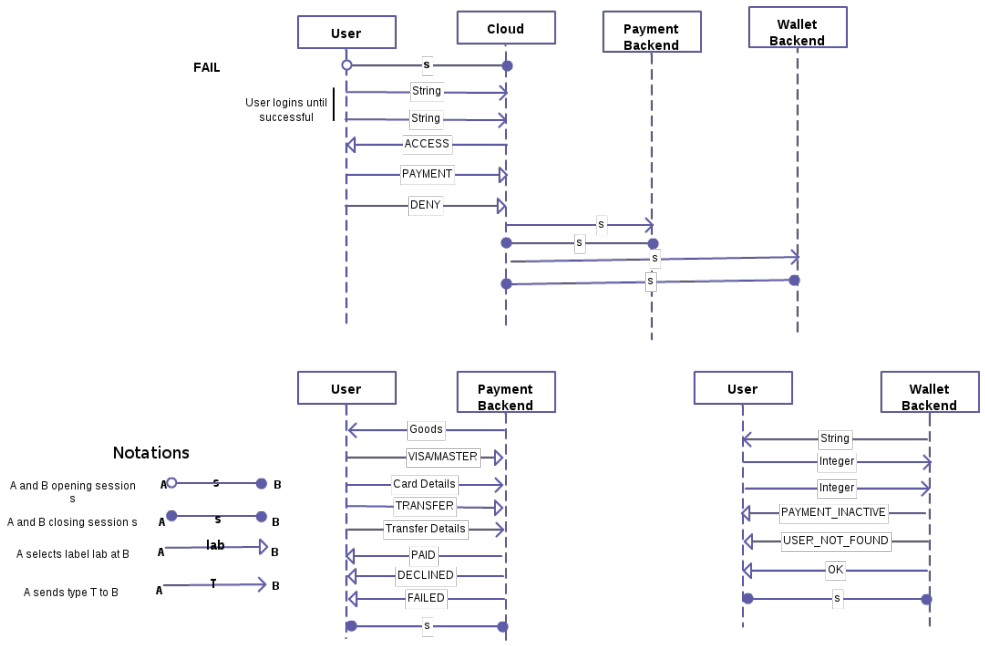

Figure 4.6: Choreography for Scenario # 3

Table 4.7: Cloud backends protocols: Payment, Wallet

Payment Backend	**Wallet Backend**
`private protocol p_payment { !<Goods>.?{ VISA_MASTER: ?(CardDetails), TRANSFER: ?(TransferDetails) }.!{ PAID: !<String>, DECLINED: !<String>, FAILED: !<String> } } private protocol p_pv { begin.?(String).?( @p_payment ) }`	`private protocol p_wallet { !<String>.?(Integer).?(Integer).!{ PAYMENT_INACTIVE: !<OSMPMessage>, USER_NOT_FOUND: !<OSMPMessage>, OK: !<OSMPMessage> } } private protocol p_wv { begin.?(String).?( @p_wallet ) }`

Interactions. After the delegation of session s is performed, the interaction of User-PaymentBackend and User-WalletBackend is provided on separate pictures of figure 4.6.

Implementation. Although the scenario is quite trivial, the tricky part is actually establishing the session between two parties. All interactions are

passed into parallel thread using the *spawn* operation. Session threads extend the *sj.runtime.net.SJThread* class. Like *java.lang.Thread*, SJThread must declare a single public void run method containing the thread body.

```
1 s_pu>.spawn(new PaymentTransactionThread(username));
2 <s_wu>.spawn(new WalletRechargeTransactionThread(username));
```

For the full code, refer to [22].

These scenarios are an example of a business offering web services similar to the way many businesses operate. These scenarios has implemented using the concepts of Session-based programming in Object-Orientated programming through the original implementation by Raymond Hu.

In different scenarios, the Master thesis underlines the process of implementing a scenario in stages.

1. Beginning from the design stage, the programmer must first consider how interactions between all parties participating in the scenario should be formed.
2. Next the protocols have to be defined accordingly, with attention given to the duality of each session.
3. By implementing the protocols the main skeleton of the program will have been completed
4. Any local computations and other properties of the programs not regarding the sessions can then be coded.

As the scenarios increase in size and complexity, directly implementing the protocols after we have defined them might be an awesome task. It would be quite helpful if at first the programmer deals with trivial protocols, to achieve compilation of the program with all the sessions set up, and only then moving on to implement the correct protocols. I tried to show that any set of interactions between two parties can be directly transformed into an SJ protocol. Mastering this is an important basis for a programmer wishing to move on with Session-based programming.

An effort was also given into using as many of the SJ session types as possible. Some examples are *inbranch - outbranch*, *outwhile - inwhile*, *delegation*. I tried to use these constructs in a variety of ways in order to identify any problems with the compiler (but failed to find anything worth mentioning), although SJ is still in development. A couple of minor points would be that it is currently necessary

that for any if statement we include an else, regardless of whether it is empty or not. Otherwise the compiler will bring an error message.

5. Evaluation

The current implementation of SJ supports all the features presented in this paper, including implementations of both the forwarding and resending protocols called *SJFSocket* and *SJRSocket*. This section, first, presents performance measurements for the current SJ runtime implementation, focusing on micro benchmarking of session initiation and the session communication primitives, then I conclude the evaluation of SJ compile and runtime from quality point of view. These preliminary results demonstrate the feasibility of session-based communication and the SJ runtime architecture.

The benchmark applications measure the time to complete a simple two-party interaction: the protocols respectively implemented by the "Server" and "Client" sides of the interaction are

$$\text{begin.?}[! < \text{MyObject} >] * \text{begin. } [?(\text{MyObject})] \tag{5.1}$$

which specify that the Server will repeatedly send objects of type *MyObject* for as long as the Client wishes. Full source code is available at [22].

Socket serves as the base case (i.e. direct usage of the underlying transport) for comparison with the SJ sockets. For the RMI implementation (RMI), the session iteration is simulated by making consecutive calls to remote server method with signature {MyObject getMyObject(boolean b)} (the boolean is passed to attain the same communication pattern). Henceforth, a session of length n means that the session-iteration is repeated n times; for RMI, n remote calls.

The benchmark applications measure the time taken for the Client to initiate a session with the Server and finish the session-iteration. For Socket, session initiation simply means establishing a connection to the server. For RMI, the connection is established implicitly by the first remote call (RMI "reuses" a server connection for subsequent calls), but we do not include the cost of looking up the remote object in the RMI registry. The RMI dummy run calls an instance of the remote object hosted on the local machine, to avoid creating a connection to the Server before the actual benchmark run.

The benchmarks were conducted using MyObject messages of serialized size

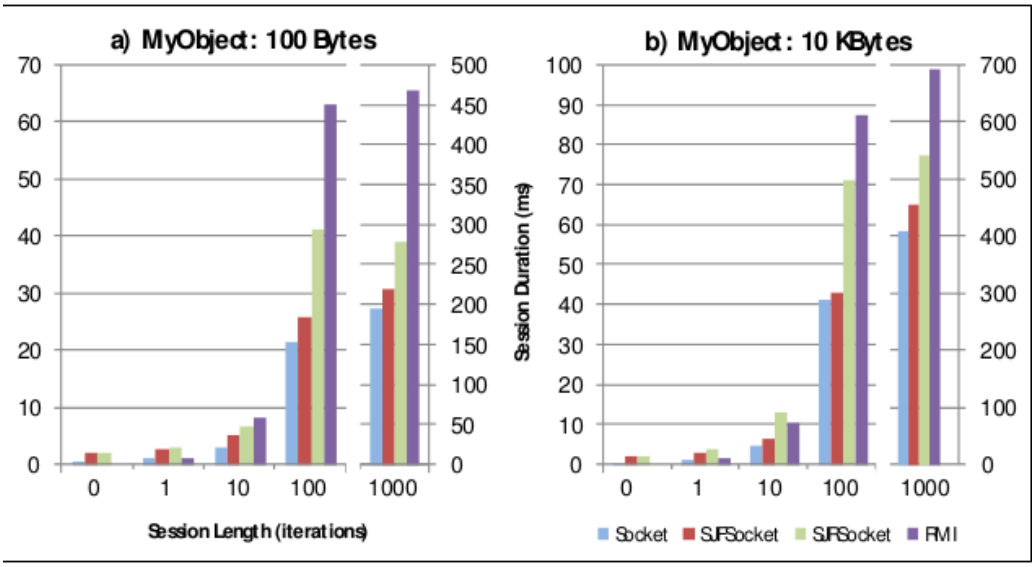

Figure 5.1: Benchmarking

100 Bytes (for reference, an Integer serializes to 81 Bytes) and 10 KBytes for sessions of length 0, 1, 10, 100 and 1000. The results are recorded from repeating each benchmark configuration 1000 times in low (about 0.1ms) and higher latency (about 10ms) environments. RMI was run using the default settings for each platform.

The low latency environment consisted of two physically neighbouring PCs (Intel Intel Core i5 1.8 GHz, 4 GB RAM) connected via gigabit Ethernet, running Ubuntu 11.04 with Java compiler and runtime (Standard Edition) version 1.7.1.

Let first look at the low latency results: The graphs a) and b) in Figure 5.1 compare the results from the four benchmark applications over each session length for each of the two MyObject sizes: using pure Socket, SJFSocket, SJRSocket and RMI. The results show that SJFSocket exhibits low runtime overhead in relation to Socket, decreasing for longer sessions. It is important to note that session initiation for both Socket and the session sockets involve establishing a TCP connection, but the SJ sockets do extra work to check session compatibility. SJRSocket is slower than SJFSocket: the cost for session initiation is the same for both, but SJRSocket employs

1. runtime state tracking and
2. a different routine for serialization and communication.

In fact, comparison of SJFSocket and SJRSocket using sessions that communicate only primitive data types [22] show that most of the overhead comes from 2nd item with runtime state tracking incurring little overhead. Despite these

additional overheads, SJRSocket performs better than RMI for longer sessions.

The results from the higher latency benchmarks [22] also support these observations. We note a few additional points. The cost of session initiation, which involves sending an extra message, increases accordingly. As before, the relative overheads of the SJ sockets become smaller for longer sessions. Indeed, the differences in performance over the longer sessions are minimal.

6. Conclusion

There is a strong need to develop structured, higher-level and type-safe abstractions and techniques for programming communications and interaction. This dissertation has presented a critical overview of web services standards and technologies existing today; deep analyzed the main issues inside the complex communication behaviour in a Web environment; introduced a pair of typed formalisms for interaction, one based on global descriptions and another an applied π-calculus, both using an extension of session types; implemented three scenarios from business environment offering web services in SJ; and compared the Session-Java performance with Java RMI and raw Java Sockets.

Global descriptions have been practiced in various engineering contexts for a long time. The presented work demonstrated its potential as a mathematically well-founded programming method, centring on type structures for communication. The presented formal theory of End-Point Projection helps to reach faultless behaviour of web services by ensuring us in well-formedness and conformance concepts.

Communication safety of distributed applications is guaranteed through a combination of static and dynamic type validation. My future work includes detailed analysis of how session-based programming can impact on the development of more complicated and large-scale applications (high-load web applications).

Preliminary benchmark results demonstrate the feasibility of session-based communication and session runtime architecture of SJ. Another interesting direction include the incorporation of different transports other than TCP into the session runtime (HTTP, HTTPs, SMTP, etc.).

The Master dissertation outcomes is a basis of the investigation of many further topics:

1. performance optimization based on asynchronous I/O
2. C10K problem[1]
3. integration to transports with order preservation message delivery [24] or unambiguous unordered delivery [25].

[1]Ability of web servers to handle ten thousand simultaneously connected clients. More can be found http://www.kegel.com/c10k.html

A. Scenario 1

Listing A.1: Cloud.sj

```
1  //$ bin/sjc tests/src/thesis/Cloud.sj -d tests/classes/
2  //$ bin/sj -cp tests/classes/ thesis.V3na 9999
3
4  package thesis;
5
6  import sj.runtime.*;
7  import sj.runtime.net.*;
8
9
10 import java.io.IOException;
11
12 import java.util.*;
13 import java.net.UnknownHostException;
14 import org.json.simple.JSONValue;
15 import org.json.simple.parser.*;
16
17
18
19 public class Cloud {
20   public static final int SUCCESS = 1;
21   public static final int UNSUCCESS = 0;
22
23   public static void main(String []argv) {
24     try {
25       new Cloud(Integer.parseInt(argv[0]), argv[1], Integer.parseInt(argv[2]));
26     } catch(IOException ioe) {
27       ioe.printStackTrace();
28     }
29
30   }
31   private protocol http_req_rep {
32       !<HttpRequestJSONMessage>.?(HttpResponseJSONMessage)
33   }
34
35   private protocol p_vu {
36     begin.?(JSONMessage).!{
37       OK: !<JSONMessage>.!<int>,
38       FAIL:
39     }
40   }
41
42   private void print(String s) {
```

```
43      System.out.println(s);
44    }
45    private boolean verify_msg(JSONMessage info) {
46      return true;
47    }
48    public Cloud(int portN, String saas_hname, int saas_port) throws IOException {
49      SJServerSocket v3naS_vu = SJRServerSocket.create(p_vu, portN);
50      SJSocket user_vu = null;
51
52      protocol p_vs { begin.@http_req_rep }
53
54      try(user_vu) {
55        user_vu = v3naS_vu.accept();
56        JSONMessage request_info = user_vu.receive();
57        print("Request for connection: " + request_info);
58        if(this.verify_msg(request_info)) {
59          user_vu.outbranch(OK) {
60            Map msg = new LinkedHashMap();
61            msg.put("status", "3");
62            msg.put("message", "Wait please.");
63            user_vu.send(JSONMessage.create(JSONValue.toJSONString(msg)));
64
65
66            SJServerAddress addr_vs = SJServerAddress.create(
67                p_vs, saas_hname, saas_port);
68            SJSocket s_vs = SJRSocket.create(addr_vs);
69            Map http_resp = null;
70            try(s_vs) {
71                s_vs.request();
72                s_vs.send(new HttpRequestJSONMessage(request_info.toString()));
73                http_resp = ((HttpResponseJSONMessage) s_vs.receive()).parse();
74            } catch(UnknownHostException uhe) {
75                uhe.printStackTrace();
76            }
77            user_vu.send(Integer.parseInt((String) http_resp.get("status")));
78          }
79        } else {
80          user_vu.outbranch(FAIL) {
81            System.out.println("CONNECTION FAILED");
82          }
83        }
84      } catch (SJIOException ioe) {
85          ioe.printStackTrace();
86      } catch(SJIncompatibleSessionException stise) {
87          stise.printStackTrace();
88      } catch(ClassNotFoundException cnfe) {
89          cnfe.printStackTrace();
90      }
```

```
91    }
92 }
```

Listing A.2: Saas.sj

```
1  //$ bin/sjc tests/src/thesis/Cloud.sj -d tests/classes/
2  //$ bin/sj -cp tests/classes/ thesis.V3na 9999
3
4  package thesis;
5  import sj.runtime.*;
6  import sj.runtime.net.*;
7
8  import java.io.IOException;
9  import java.util.*;
10 import org.json.simple.JSONValue;
11 import org.json.simple.parser.*;
12
13 class Saas {
14     private static final int OK = 1;
15     public static void main(String []argv) {
16         try {
17             new Saas(Integer.parseInt(argv[0]));
18         } catch(IOException ioe) {
19             ioe.printStackTrace();
20         }
21
22     }
23     private protocol p_sv {
24         begin.?(HttpRequestJSONMessage).!<HttpResponseJSONMessage>
25     }
26
27     public Saas(int portNumber) throws IOException{
28         SJServerSocket server_sv = SJRServerSocket.create(p_sv, portNumber);
29         SJSocket v3na_sv = null;
30         try(v3na_sv) {
31             v3na_sv = server_sv.accept();
32             HttpRequestJSONMessage msg = v3na_sv.receive();
33             Map response = new LinkedHashMap();
34             response.put("message", "Created.");
35             response.put("status", "1");
36             v3na_sv.send(new HttpResponseJSONMessage(JSONValue.toJSONString(
                   response)));
37         } catch(Exception e) {
38             e.printStackTrace();
39         }
40     }
41 }
```

Listing A.3: User.sj

```
1  //$ bin/sjc tests/src/thesis/User.sj -d tests/classes/
2  //$ bin/sj -cp tests/classes/ thesis.User localhost 9999
3
4
5  package thesis;
6
7  import sj.runtime.*;
8  import sj.runtime.net.*;
9
10 import java.util.*;
11
12 import org.json.simple.JSONValue;
13 import org.json.simple.parser.*;
14 import java.util.Date;
15 import java.sql.Timestamp;
16 // import thesis.utils.JSON;
17 public class User
18 {
19   public static void main(String []argv) {
20     try {
21
22       new User(argv[0], Integer.parseInt(argv[1]));
23
24     } catch(SJIOException sjioe) {
25
26       sjioe.printStackTrace();
27
28     }
29
30   }
31   private static protocol p_uv {
32       begin.
33       !<JSONMessage>.
34       ?{
35         OK: ?(JSONMessage).?(int),
36         FAIL:
37       }
38   }
39
40   private Map buildConnectionRequest() {
41       long curTime = new Timestamp((new Date()).getTime()).getTime();
42       Map msg = new LinkedHashMap();
43       msg.put("action", "CONNECTION");
44       msg.put("client_email", "r.kamun@gmail.com");
45       msg.put("ts", curTime + "");
46       return msg;
47   }
```

```
48    private void print(String s) {
49        System.out.println(s);
50    }
51
52    public User(String hname, int port) throws SJIOException {
53        SJServerAddress v3na_addr = SJServerAddress.create(p_uv, hname, port);
54        SJSocket s_uv = SJRSocket.create(v3na_addr);
55
56        try(s_uv) {
57          s_uv.request();
58
59          Map msg = this.buildConnectionRequest();
60          s_uv.send(JSONMessage.create(JSONValue.toJSONString(msg)));
61          msg = null;
62
63          s_uv.inbranch() {
64            case OK: {
65              msg = ((JSONMessage) s_uv.receive()).parse();
66              print("Status: " + msg.get("message"));
67              int status = s_uv.receiveInt();
68              print("Status: " + status);
69            }
70            case FAIL: {
71              print("FAILED");
72            }
73          }
74        } catch(SJIOException sjioe) {
75            sjioe.printStackTrace();
76        } catch(Exception e) {
77            e.printStackTrace();
78        }
79    }
80 }
```

B. Scenario 2

Listing B.1: Cloud.sj

```
1  //$ bin/sjc tests/src/thesis/Cloud.sj -d tests/classes/
2  //$ bin/sj -cp tests/classes/ thesis.V3na 9999
3
4  package thesis.scenario2;
5
6  import sj.runtime.*;
7  import sj.runtime.net.*;
8
9
10 import java.io.IOException;
11
12 import java.util.*;
13 import java.net.UnknownHostException;
14 import org.json.simple.JSONValue;
15 import org.json.simple.parser.*;
16
17
18
19 public class Cloud {
20   public static final int SUCCESS = 1;
21   public static final int UNSUCCESS = 0;
22   private Map database = new LinkedHashMap();
23   public static void main(String []argv) {
24     try {
25       new Cloud(Integer.parseInt(argv[0]), argv[1], Integer.parseInt(argv[2]));
26     } catch(IOException ioe) {
27       ioe.printStackTrace();
28     }
29
30   }
31   private protocol p_vs {
32       begin.
33       !< !{
34             OK: !<JSONMessage>,
35             FAIL: !<JSONMessage>
36       } >.!<JSONMessage>
37   }
38
39   private protocol p_vu {
40     begin.
41     ![
42       ?(String).?(String)      // login password
```

```
43      ]*.
44        !{
45          ACCESS: ?(JSONMessage).
46            !{
47              OK: !<JSONMessage>,
48              FAIL: !<JSONMessage>
49            },
50          DENY: !<String>
51        }
52    }
53
54    private void print(String s) {
55       System.out.println(s);
56    }
57    private boolean verify_msg(JSONMessage info) {
58      return true;
59    }
60    private boolean is_authenticated(String login, String password) {
61        try {
62            return this.database.get(login).equals(password);
63        } catch(java.lang.NullPointerException exc) {
64            return false;
65        }
66    }
67    private void connectToDB() {
68        this.database.put("r.kamun@gmail.com", "00112358");
69    }
70    public Cloud(int portN, String saas_hname, int saas_port) throws IOException {
71      this.connectToDB();
72
73      SJServerSocket v3naS_vu = SJRServerSocket.create(p_vu, portN);
74      SJSocket user_vu = null;
75
76      try(user_vu) {
77        user_vu = v3naS_vu.accept();
78        boolean exit = false;
79        int counter = 0, max_atempts = 5;
80        user_vu.outwhile(!exit) {
81            String login = user_vu.receive();
82            String password = user_vu.receive();
83            counter ++;
84            if(this.is_authenticated(login, password) || (counter >= max_atempts))
                  {
85                exit = true;
86            }
87        }
88        if(counter < max_atempts) {
89            user_vu.outbranch(ACCESS) {
```

```
90                JSONMessage req_info = user_vu.receive();
91                SJServerAddress addr_vs = SJServerAddress.create(
92                    p_vs, saas_hname, saas_port);
93                SJSocket s_vs = SJRSocket.create(addr_vs);
94                try(s_vs) {
95                    s_vs.request();
96                    s_vs.send(user_vu);    // pass the remaining protocol
97                    s_vs.send(req_info);
98                } catch(UnknownHostException uhe) {
99                    uhe.printStackTrace();
100               }
101           }
102       } else {
103           user_vu.outbranch(DENY) {
104               user_vu.send("You have no permissions. BYE!");
105           }
106       }
107
108   } catch (SJIOException ioe) {
109       ioe.printStackTrace();
110   } catch(SJIncompatibleSessionException stise) {
111       stise.printStackTrace();
112   } catch(ClassNotFoundException cnfe) {
113       cnfe.printStackTrace();
114   }
115 }
116 }
```

Listing B.2: Saas.sj

```
1  //$ bin/sjc tests/src/thesis/Cloud.sj -d tests/classes/
2  //$ bin/sj -cp tests/classes/ thesis.V3na 9999
3
4  package thesis.scenario2;
5  import sj.runtime.*;
6  import sj.runtime.net.*;
7
8  import java.io.IOException;
9  import java.util.*;
10 import org.json.simple.JSONValue;
11 import org.json.simple.parser.*;
12
13 class Saas {
14     private static final int OK = 1;
15     public static void main(String []argv) {
16         try {
17             new Saas(Integer.parseInt(argv[0]));
18         } catch(IOException ioe) {
19             ioe.printStackTrace();
```

```
20          }
21
22      }
23      private protocol p_msg {
24          !{
25            OK: !<JSONMessage>,
26            FAIL: !<JSONMessage>
27          }
28      }
29
30      private protocol p_sv {
31          begin.?(@p_msg).?(JSONMessage)
32      }
33      private boolean verify_msg(JSONMessage params) {
34          return 1 == 1;
35      }
36      private boolean validate_msg(JSONMessage params) {
37          return true;
38      }
39      private JSONMessage genResponse(String status, String message) {
40          Map m = new LinkedHashMap();
41          m.put("status", status);
42          m.put("message", message);
43          return JSONMessage.create(JSONValue.toJSONString(m));
44      }
45      public Saas(int portNumber) throws IOException{
46          SJServerSocket server_sv = SJRServerSocket.create(p_sv, portNumber);
47          SJSocket v3na_sv = null;
48          SJSocket v3na_user_socket = null;
49          try(v3na_sv, v3na_user_socket) {
50              v3na_sv = server_sv.accept();
51              v3na_user_socket = (@p_msg) v3na_sv.receive();
52              JSONMessage req_params = (JSONMessage) v3na_sv.receive();
53              boolean allowed = this.verify_msg(req_params) && this.validate_msg(
                    req_params);
54              if(allowed) {
55                  v3na_user_socket.outbranch(OK) {
56                      v3na_user_socket.send(
57                          this.genResponse("1", "USER CREATED. Email has been sent
                                .")
58                      );
59                  }
60              } else {
61                  v3na_user_socket.outbranch(FAIL) {
62                        v3na_user_socket.send(this.genResponse("0", "FAILED."));
63                  }
64              }
65          } catch(Exception e) {
```

```
66              e.printStackTrace();
67          }
68      }
69  }
```

Listing B.3: User.sj

```
1  package thesis.scenario2;
2
3  import sj.runtime.*;
4  import sj.runtime.net.*;
5
6  import java.util.*;
7
8  import org.json.simple.JSONValue;
9  import org.json.simple.parser.*;
10 import java.util.Date;
11 import java.sql.Timestamp;
12 import java.io.*;
13 public class User
14 {
15   public static void main(String []argv) {
16     try {
17
18       new User(argv[0], Integer.parseInt(argv[1]));
19
20     } catch(SJIOException sjioe) {
21
22       sjioe.printStackTrace();
23
24     }
25
26   }
27   private static protocol p_uv {
28       begin.?[
29         !<String>.!<String>
30       ]*.
31         ?{
32           ACCESS: !<JSONMessage>.
33             ?{
34                 OK: ?(JSONMessage),
35                 FAIL: ?(JSONMessage)
36             },
37           DENY: ?(String)
38       }
39   }
40
41   private Map buildConnectionRequest() {
42       long curTime = new Timestamp((new Date()).getTime()).getTime();
```

```
43          Map msg = new LinkedHashMap();
44          msg.put("action", "CONNECTION");
45          msg.put("client_email", "r.kamun@gmail.com");
46          msg.put("ts", curTime + "");
47          return msg;
48      }
49      private void print(String s) {
50          System.out.println(s);
51      }
52
53      private String login;
54      private String password;
55      Scanner sc = new Scanner(new InputStreamReader(System.in));
56      public User(String hname, int port) throws SJIOException {
57          SJServerAddress v3na_addr = SJServerAddress.create(p_uv, hname, port);
58          SJSocket s_uv = SJRSocket.create(v3na_addr);
59
60          try(s_uv) {
61            s_uv.request();
62            // login password
63            s_uv.inwhile() {
64              System.out.println("Enter your login:");
65              this.login = sc.nextLine();
66              System.out.println("Enter your password:");
67              this.password = sc.nextLine();
68              s_uv.send(this.login); s_uv.send(this.password);
69            }
70            s_uv.inbranch() {
71                case ACCESS: {
72                    s_uv.send(
73                        JSONMessage.create(
74                            JSONValue.toJSONString(this.buildConnectionRequest())
75                        )
76                    );
77                    s_uv.inbranch() {
78                      case OK: {
79                        Map msg = ((JSONMessage) s_uv.receive()).parse();
80                        print("Result: " + msg.get("message"));
81                      }
82                      case FAIL: {
83                        Map msg = ((JSONMessage) s_uv.receive()).parse();
84                        print("Reason: " + msg.get("message"));
85                      }
86                    }
87                }
88                case DENY: {
89                    print((String)s_uv.receive());
90                }
```

```
91            }
92
93        } catch(SJIOException sjioe) {
94            sjioe.printStackTrace();
95        } catch(Exception e) {
96            e.printStackTrace();
97        }
98    }
99 }
```

Bibliography

[1] Matjaz B. Juric, Ramesh Loganathan, Poornachandra Sarang, and Frank Jennings. *SOA Approach to Integration*. PACKT, Birmingham — Mumbai, 2007.

[2] Steve Ross-Talbot and Tony Fletcher. Ws-cdl primer. *W3C*, 2006.

[3] C. Hoare. *Communicating Sequential Processes*. Prentice Holl, 1985.

[4] H. Raymond, Y. Nobuko, and H. Kohei. Session-based distibuted programming in java. *Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 516–541, 2008.

[5] Rabbitmq, http://rabbitmq.com.

[6] Zeromq, zeromq.org.

[7] Soap: Messaging framework, http://w3.org/TR/soap12-part1/, 2007.

[8] Wsdl specification, http://w3.org/TR/wsdl.

[9] Rest (representational state transfer), http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[10] A. Hoing. Orchestrating secure workflows. *PhD Dissertation*, 2010.

[11] B. Alistair, D. Marlon, and O. Phillipa. A critical overview of the web service choreography description language (ws-cdl). *BPTrends*, 2005.

[12] Z. Qiu, C. Cai, X. Zhao, and H. Yang. Exploring the essence of choreography. *Proceedings of the 16th international conference on World Wide Web*, pages 973–982, 2007.

[13] M. Carbone, H. Kohei, and Y. Nobuko. Structured communication-centered programming for web services. *ACM Transactions on Programming Languages and Systems*, 34, 2012.

[14] A. Abdallah, C. Jones, and J. Sanders. *Communication Sequential Processes: The first 25 years.*

[15] M. Carbone. A theoretical basis of communication-centered concurrent programming. *ACM New York*, 2012.

[16] K. Honda and V. T. Vasconcelos. Language primitives and type discipline for structured. *ESOP'98*, pages 122–138, 1998.

[17] K. Honda and V. T. Vasconcelos. Control in the p-calculus. *ACM-SIGPLAN Continuation*, 2004.

[18] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[19] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, pages 409–453, 1996.

[20] R. Milner. *The polyadic p-calculus.* Springer-Verlag, 1993.

[21] H. Raymond, Y. Nobuko, B. Andi, and H. Kohei. The sj framework for transport-independent, type-safe, object-oriented communications programming. *CiteSteer.*

[22] Rustem Kamun. Master-thesis, https://github.com/Rustem/Master-thesis.

[23] Rustem Kamun. E-commerce cloud http://v3na.com, 2013.

[24] E. Kohler and S. Floyd. Designing dccp: congestion control without reliability. *SIGCOMM Comput. Commun*, pages 27–38, 2006.

[25] Advanced message queueing protocols (amqp), http://jira.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Protocol.

MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

JSC "Kazakh-British Technical University"
Faculty of Information Technologies

SCIENTIFIC PUBLICATIONS

by Information Systems M.Sc. programme student Rustem Kamun

No.	Title	Publisher	Pages	Co-authors
1	Smart Shopping Cart	KBTU, Сборник трудов, Международный Конкурс студенческих проектов по информационным технологиям Алматы, 28 апреля, 2012	153-159	
2	Synergy of Service-Oriented Architecture and Cloud Computing Introduction	KBTU, Сборник трудов, Международный Конкурс студенческих проектов по информационным технологиям Алматы, 2013		

Author

Rustem Kamun

Scientific secretary

MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

JSC "Kazakh-British Technical University"
Department of Computer Engineering

Dissertation supervisor review
by Timur Umarov

on "Ensuring faultless web services specifications by developing discrete and modular systems blueprints"

6M070300 – Information Systems

Rustem A. Kamun have made a solid work by researching business processes and their interaction in Web environment. I am sure that Rustem Kamun's result is very valuable and actual for the Internet computing and Web services.

He carefully analyzed the issues of existing technologies, the issues of existing standards like WS-CDL and BPEL-WS and proposed the formal theory based on session typing.

Rustem A. Kamun confirmed the feasibility of the proposed theory by applying it in designing business protocols. He also received significant results by comparing the performance of Session Java with existing technologies.

It's worth noting that Rustem's work has a huge impact on Web-service analysis and made a base for his future Ph.D. I recommend to evaluate the dissertation of Rustem A. Kamun as "excellent" and award him with a deserved qualification "MSc in Information Systems" by specialty 6M070300.

Scientific supervisor

Timur F. Umarov,
Ph.D. Computer Science,
Associate professor

"____" _____ 2013