Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado em Engenharia Informática

Service-oriented Mobility of Java Code in Web Services-based Architectures

Gilberto Camacho (30189)

Lisboa 2010

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado em Engenharia Informática

Service-oriented Mobility of Java Code in Web Services-based Architectures

Gilberto Camacho (30189)

Orientador: Prof. Doutor Hervé Paulino

Dissertação apresentada na Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa para a obtenção do Grau de Mestre em Engenharia Informática

Lisboa 2010

To my dear parents Fernando and Maria Camacho

Acknowledgements

Before proceeding with the introduction of my thesis, I would like to thank all the people involved with me that in one way or another, have contributed to this experience in becoming successful and unique.

I would like to thank:

- My parents Fernando Camacho and Maria Camacho, and brother José Camacho for their continuous support throughout the course.
- My friends Miguel Teixeira and Ricardo Palha who have helped me in the last days of the evaluation of the system.
- My colleagues Danilo Manmohanlal, Paulo Mariano and Amavel Santo who have shared knowledge (and much coffee) to overcome difficulties in the implementation period.
- At last but not the least, my advisor and Professor Hervé Paulino who has proposed this dissertation, for his continuous support and encouragement since day one with much patience.

Resumo

A mobilidade de software é uma tecnologia que consiste em fornecer mobilidade a componentes de software para que possam migrar para um computador remoto para interagir localmente. Por outras palavras, esta tecnologia permite que a computação seja transferida de uma máquina para outra remota. Ao fornecermos esta capacidade de transferir computações entre diferentes máquinas, é de esperar que surjam preocupações relacionadas com a segurança. Por agora, acreditamos que o paradigma da mobilidade de software limita-se a ambientes com bases de confiança, tais como redes locais ou camadas de middleware, onde as questões de segurança podem ser melhor controladas.

A computação orientada a serviços reorganiza a arquitectura de rede na forma de serviços, onde os seus componentes são mais facilmente integrados, modificados ou removidos. Eles têm a capacidade de cooperar entre si, independentemente da linguagem de programação utilizada no seu desenvolvimento. Além disso, a computação orientada a serviços é uma tecnologia amplamente aceite para a implementação de aplicações distribuídas como por exemplo, o middleware.

O trabalho realizado nesta tese consiste em instanciar um modelo que combine mobilidade de software e computação orientada a serviços, tal como foi proposto por Paulino [20]. Neste modelo, as sessões de migração tiram proveito dos recursos de uma rede orientada a serviços, criando assim um ambiente onde a migração é modelada em termos de serviços em vez de abstracções de nós de rede.

Neste trabalho, pretendemos aplicar a migração de programas Java no contexto de uma arquitectura orientada a serviços desenvolvidos com Web services. Esta aplicação é composta por uma camada de middleware que corre entre o programa fonte e os serviços de tecnologias Web, e cuja interface é o resultado do mapeamento das operações especificadas no modelo.

A avaliação efectuada ao modelo instanciado permitiu-nos identificar situações em que a migração do componente para o servidor para interagir localmente é mais vantajosa comparativamente com a interacção remota com o mesmo.

Abstract

Software mobility consists of providing software components, the ability to migrate to a remote host with the purpose of interacting locally. In other words, this technology enables computations to be transferred from the current machine to a remote one. This powerful enhancement embodied in a traditional network fairly raises security concerns. For now, we believe that software mobility paradigm is confined to environments with bases of trust such as local area networks or middleware layers where security issues can be better controlled.

Service-oriented computations reorganize the network architecture in the form of services, where components are more easily integrated, modified and removed. They have the ability to cooperate between them regardless the programming language used in their development. In addition, service-oriented computing is a widely accepted technology for the implementation of distributed applications, namely middleware.

The work developed in this thesis consists of instantiating a model which combines software mobility and service-oriented paradigms as proposed by Paulino [20]. In this model, migrating sessions take advantage of the resources of a service-oriented network, creating thus an environment where the migration is modeled in terms of services instead of network nodes abstractions.

In the instantiated model, we aim to apply the migration of Java programs in a context of a service-oriented architecture developed with Web services. This application comprises of a middleware layer that runs between the source program and the Web services technologies, and whose interface is the result of the mapping of the operations defined in the model.

The evaluation performed to the instantiated model allows us to identify situations in which component migration to the server to interact locally is more advantageous in comparison to remote interacting with it.

Contents

1. Introduction 1

1.1. Motivation 1

1.2. Problem statement and work goals 2

1.3. Proposed solution 3

1.4. Contributions 3

1.5. Document Outline 4

2. State-of-the-art 5

2.1. Software Mobility 5

2.1.1. The Software Mobility Paradigm 5

2.1.2. Mobility Models 6

2.1.3. Advantages of Software Mobility 7

2.1.4. Disadvantages of Software Mobility 8

2.1.5. Applications 8

2.1.6. Execution Support 9

2.1.7. Systems Supporting Software Mobility 11

2.2. Service-oriented paradigm 12

2.2.1. Distributed Objects Architecture 13

2.2.2. Service-Oriented Architecture 14

2.2.3. Web Service Protocols and Technologies 15

3. Service-Oriented Software Mobility 19

3.1. Overview 19

3.2. Itinerary 21

3.3. Unidirectional traveling 22

3.4. Multidireccional traveling 23

3.5. Bridging 24

4. Instantiation of the Service-Oriented Mobility Model 27
4.1. Overview 27
4.1.1. Scenarios 28
4.1.2. Data Manager 29
4.2. User Application Programming Interface 30
4.2.1. Server 30
4.2.2. Result 31
4.2.3. Session 31
4.2.4. Data Manager 33
4.2.5. Itinerary 34
4.2.6. Bridging 38
4.2.7. Code sample 40
4.3. Middleware Architecture 43
4.3.1. Session Setting-up 44
4.3.2. Session Execution 45
4.3.3. Session Management 45
4.3.4. Communication 45
4.4. Itinerary Lifecycle 46
4.5. Middleware Implementation 47
4.5.1. User-developed components transformation into middleware components 47
4.5.2. Itinerary departure procedures 50
4.5.3. Itinerary server arrival procedures 53
4.5.4. Results returning to client 55
5. Evaluation 57
5.1. Environment Specifications 58
5.2. Measuring the overhead of the middleware 58
5.3. Remote interaction versus remote execution (lower bandwidth) 59
5.3.1. Remote interaction 59
5.3.2. Remote execution 60
5.3.3. Comparison between remote interaction and remote execution 62
5.4. Remote interaction versus remote execution (higher bandwidth) 63
5.4.1. Comparison between remote interaction and remote execution 63
5.5. Interactions 64

5.5.1. Remote interaction versus remote execution (lower bandwidth) 64
5.5.2. Remote interaction versus remote execution (higher bandwidth) 67
5.6. Unidirectional...68
5.6.1. Remote interaction versus remote execution (lower bandwidth) 70
5.6.2. Remote interaction versus remote execution (higher bandwidth) 73
5.7. Multidirectional (Speed-Up) 76
6. Conclusions 78
6.1. Summary 78
6.2. Future work 79
7. Bibliography...80

List of Figures

1.1 - General overview of the system architecture 3
2.1 - Strong mobility 6
2.2 - Weak mobility 6
2.3 - Find-bind-execute paradigm 15
2.4 - Organization of a Web service through WSDL format 17
3.1.1 - Session uploading in a service-oriented system 21
3.2.1 - Session migrating to two service' providers 21
3.3.1 - Upload with unidirectional traveling strategy applied 23
3.4.1 - Upload with multidirectional traveling strategy applied 24
3.5.1 - Manual bridging 25
3.5.2 - System bridging 26
4.1.1 - General overview of the system 27
4.1.2 - One-to-one scenario 28
4.1.3 - One-to-many scenario 29
4.1.4 - Many-to-many scenario 29
4.2.1 - A session interacting with several data managers 34
4.2.2 - Example of an upload with both traveling strategies combined 35
4.2.3 - Session using a local data manager to store results 37
4.2.4 - Session using a remote data manager to store results 37
4.2.5 - Bridging between two sessions 38
4.2.6 - Bridge in unidirectional traveling 39
4.2.7 - Bridge in multidirectional traveling 40
4.3.1 - Middleware architecture 44
4.4.1 - Lifecycle of an itinerary 46
4.5.1 - Session and Itinerary contents in the client and in the middleware 48
4.5.2 - Middleware itinerary 50
4.5.3 - Example of an itinerary before leaving the client machine 52
4.5.4 - Real traveling route of itinerary 117 54
4.5.5 - Handler synchronization 56
5.1 - Session executing in the server and interacting in the server 57
5.2 - Session executing in the client and interacting with a remote server 58
5.3.1 - Graph of Table 5.3.3 62
5.4.1 - Graph of Table 5.4.1 63
5.5.1 - Graph of Table 5.5.3 66
5.5.2 - Graph of Table 5.5.4 67

5.6.1 - Number of classes transferred from host to host (3 servers) 68
5.6.2 - Graph of table 5.6.3 72
5.6.3 - Graph of table 5.6.6 75
5.7.1 - Graph of table 5.7.1 77

List of Tables

5.2.1 - Average time values in session uploading 58
5.3.1 - Remote interaction time values (1Mbps) 60
5.3.2 - Remote execution time values (1Mbps) 61
5.3.3 - Remote interaction versus remote execution (1Mbps) 62
5.4.1 - Remote interaction versus remote execution (100Mbps) 63
5.5.1 - Remote interacting with interactions (1Mbps) 64
5.5.2 - Remote executing with interactions (1Mbps) 65
5.5.3 - Remote interaction versus remote execution with interactions (1Mbps) 66
5.5.4 - Remote interaction versus remote execution with interactions (100 Mbps) 67
5.6.1 - Unidirectional remote execution (1Mbps) 70
5.6.2 - Unidirectional remote interaction (1Mbps) 71
5.6.3 - Unidirectional remote execution versus remote interaction (1Mbps) 72
5.6.4 - Unidirectional remote execution (100Mbps) 73
5.6.5 - Unidirectional remote interaction (100Mbps) 74
5.6.6 - Unidirectional remote execution versus remote interaction (100Mbps) 75
5.7.1 - Multidirectional speed-up (1Mbps) 76

1. Introduction

1.1. Motivation

Software mobility paradigm is a well-studied technology focused on the development of distributed components. Environments supplying this kind of technology allow the same computation to run in any host across the network. The contribution given by software mobility in the computers science field is particularly interesting as it provides components to be programmed in one machine to migrate to a remote one in order to be executed. Java applet is an example of an application that benefits from code mobility for its execution. Notwithstanding the advantages introduced by software mobility, the potential existing in permitting mobile agents to travel towards the resources raised justified concerns about security.

Considering the uneasiness related to security issues, we believe that with this state-of-the-art, the use of mobile agents are confined to environments with bases of trust such as local area networks running under a single administrative domain or systems offering a middleware layer which has control mechanisms over migrated code.

Architectures supporting service-oriented computing provide a loosely-integrated set of services that can be used within multiple domains. We found interesting in making use of service-oriented computing technology in our work since it offers abstraction of both network resources and software components. Indeed, the composition of loosely-bound service-oriented components typical in this kind of architecture has been proved to be a good paradigm for the modeling of distributed applications such as mobile agents.

Additionally, Web service is an emerging technology that fits greatly in this context since it provides communication between loosely-coupled components thanks to the "contract" offered by the WSDL (Web Services Description Language). It is our opinion that mobility has the potential to be a useful technology for the future of the Internet, especially when combined with services.

1.2. Problem statement and work goals

Nowadays, software migration is modeled in terms of network node abstractions, IP addresses or URLs (Uniform Resource Locator). In common mobile agent development APIs, applications require to know the location of the server. As such, the operational flow of a program that wants to perform the sequence of operations P1 to Pn at hosts h1 to hn is:

go to host h1 and do P1;
go to host h2 and do P2;
…
go to host hn and do Pn;

However, the possible absence of a specific host and the search for alternatives obligates the programmer to implement solutions to overcome these obstacles.

Service-oriented computing is a kind of technology that permits mobility to be modeled in terms of services rather than hosts. A migrating program takes advantage of this technology since it is defined to find an instance of a service to operate instead of being set to go to a specific host to operate. In addition, the inclusion of service-orientation removes the burden from the programmer to implement solutions to surpass the problem of a missing network node.

Thus, the operational flow of a mobile agent in this environment is:

find an instance of service S1 and do P1;
find an instance of service S2 and do P2;
…
find an instance of service Sn and do Pn;

This approach requires the search for alternatives to be in charge of a service discovery mechanism. Nonetheless, the handling of situations where there are no alternatives is still required.

In this thesis, we propose to instantiate a model in a mainstream programming language (Java) and to offer support so the migration is modeled in terms of services, namely Web services. The system consists of a middleware layer that allows multiple hosts to be connected with each other with the objective of managing user-programmed components that are uploaded to the network. Clients are able to develop mobile agents to run in this system through an API specifically designed to provide mechanisms that interact with the middleware. There are no restrictions in the code developed by the user for the session to execute remotely. Additionally, this API also allows the user to define the traveling strategy a session – or a set of sessions – must follow.

1.3. Proposed solution

Our proposal is to instantiate a model (Fig. 1.1) comprising of a middleware layer that is defined to support sessions' migration in a Web environment. This software layer is responsible in collecting the classes that sessions require in order to be executed remotely, create a Web service request, disposing a Web service in the server to receive migrating sessions, load the collected classes to the Java Virtual Machine (JVM), execute sessions, collect the computed results to return to the client, provide a Web service in the client machine to attend incoming results and synchronize the client application request with the results that have returned.

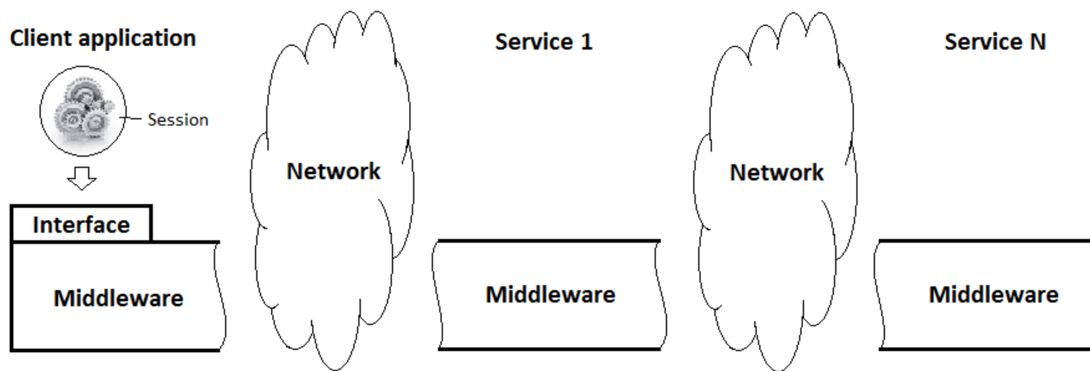


Figure 1.1 – General overview of the system architecture

In order to utilize the features provided by the middleware, the programmer has access to an application programming interface (API) that allows him/her to develop components and to dispatch them to the network. With this API, the programmer has the ability to create sessions to be executed in remote stations and to receive the respective computed results. In addition, the user can define the destinations for the session to migrate to as well as the route it should take throughout its trip in the network. The API allows the user to specify a session to travel from host to host in sequence (unidirectional) or in parallel (multidirectional).

Once the system is implemented, a study of performance will be realized. The objective of the study is to understand which are the situations that favor migrating a session instead of having it interacting remotely.

1.4. Contributions

We aim to contribute with this work, a model instantiated in Java language that offers an API for the programmer to make use of the abstract concepts of the model.

We also intend to provide a middleware in our model that features user-developed components to travel to any station in the network that is running our middleware. The user-programmed

components travel according to the route defined by the user, in sequence and/or in parallel. Additionally, these components feature interactions between them once deployed in the network, such as copying the computed results from one to another.
Finally, we pretend to offer an extensive study of impact consisting of identifying which situations favors session migration and which situations favors remote interaction. The studies vary from increasing the bandwidth from 1 Mbps to 100 Mbps, increasing the number of interactions with the server as well as the number of servers that a session interacts with.

1.5. Document Outline

The thesis is structured in six chapters. The next chapter introduces the software mobility and the service-oriented paradigms. It explains the different approaches used for systems under each paradigm. Chapter 3 describes the combination of these two paradigms: service-oriented mobility. Chapter 4 explains the instantiation of the service-oriented mobility model. It presents the API for the programmer and a detailed explanation of the middleware implementation. Chapter 5 provides the studies performed in our model with the objective of comprehending which situations favor migration and which situations favor remote interaction. And at last, Chapter 6 presents the conclusions of this thesis and future work.

2. State-of-the-art

In this chapter, we provide an in-depth lecture of Software Mobility and Service-oriented computing distributed in two sections. Section 2.1 offers a perspective of computations capable of traveling to remote computers to be executed locally and in Section 2.2 the attention is centered on service-oriented computing and in Web services. We aim to provide in these two sections, the background for Service-oriented Mobility (Chapter 3).

2.1. Software Mobility

2.1.1. The Software Mobility Paradigm

Software mobility [5, 13, 18] is a characteristic of software which has the ability to travel across multiple hosts in a network, in order to perform computation activities locally. It does not rely on remote sessions to exchange messages because all the information needed to perform its job is moved toward the resources. However, before migrating from one host to another, the software does need to be acknowledged of which computer to go. A software infrastructure running in the network is responsible for giving support to mobile software, providing such as data protection and security.

Software mobility focuses essentially on the potential that can be achieved in having software components executing remotely rather than executing in the local machine.

A mobile agent is a computer program that runs on behalf of a network user and is intended to execute all its computing operations locally. It consists of an encapsulation of code, data and process state. When a computer is connected to a network, a mobile agent can travel to a new host by halting its execution on the current machine, saving its state and restoring it back in the destination. After being in execution in the recipient host, the agent may have collected information to return back to the source host or it may travel to another remote computer to continue its work.

The autonomy evidenced by mobile agents is most visible when they migrate from one host to another. In fact, a mobile agent is intended to have contact with its source host only in two occasions: when it departs and when it returns to the starting node. Whatever happens in between, a mobile agent enjoys the freedom to be completely detached from its source host to perform its task in the distributed environment. Even if the starting node gets disconnected

from the network in this meantime, this will not bother the mobile agent because it was given all the instructions needed to perform its task successfully before departing from the source host.

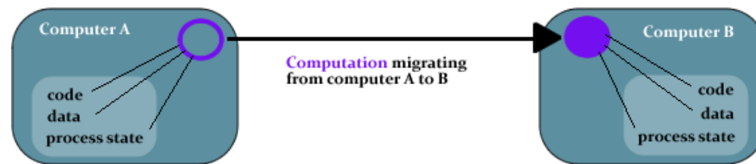


Figure 2.1 - Strong mobility

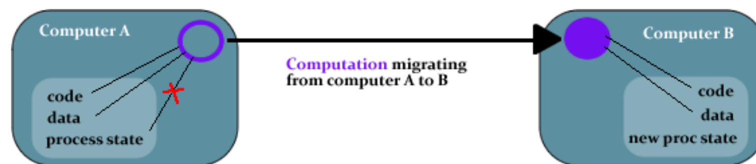


Figure 2.2 - Weak mobility

2.1.2. Mobility Models

In the process of traveling from one machine to another, a mobile agent can be classified as supporting strong or weak mobility [5]. The former comprises of code, data and process state, and the latter comprises of just code and data. Figures 2.1 and 2.2 represent the migration of an agent with strong and weak mobility, respectively. The Java Virtual Machine (JVM) is an example of a popular run-time environment used to execute Web related applications which supports only weak mobility. Due to its security defined boundaries, the JVM does allow access only to code and data and not to the run-time state of the process [21].

When using weak mobility, since the run-time state of the agent is not transferred, the content sent must be given a point of reference in order to be successfully re-instantiated in the recipient host as intended. It could for instance, be given instructions to execute starting from a specific function when arriving in the new host. The re-instantiation process is explained in more detail in subsection 2.1.6.

In software mobility, there are two types of behavior which an agent can have: proactive or reactive [5]. Proactive behavior is attributed to agents that decide themselves when and where to migrate whereas reactive behavior is attributed to agents that react to some external event that will trigger the migration.

There are two different situations which can trigger mobile code transferring [5]: code-shipping regards to when the source host sends all the code to the destination computer and code-fetching is when the destination host downloads the code from some source host. The main difference resides in which application makes the decision for the migration - if the one in the source host or the one in the destination host.

2.1.3. Advantages of Software Mobility

The use of mobile agents in distributed systems brings some advantages some of which are itemized below:

- Overcoming client computer limitations [13] - communication delays, short memory size, limited storage capacities, insufficient network bandwidth and/or low CPU processing capacity are difficulties that may be found on a client computer. The performance can be improved if an agent is sent to the recipient host to process the data locally rather than accessing it remotely;

- Survivability rate [13] - the benefit of transferring code, data and state encapsulated makes a mobile agent survivability rate higher compared to the client-server model. In fact, as longer a session has to be maintained, more costly it is likely to become for the involved hosts and by consequence, the request it carries on takes longer than expected. Mobile agents are independent entities that do not require sessions to perform their tasks. And even if a network node has failed, it will not reduce mobile agents working pace since they are intelligent enough to decide to move to a different host to continue their work.

- Customization [13] - mobile agents can be easily customized according to the user needs whereas client-server model require more time to adapt to a new environment. Given the possibility to request the remote execution of code, the server does not need to be up-graded to run new functionalities in order to attend client unforeseen needs. Therefore, memory and storage capacities, and other resources are spared from being consumed.

- Portability [18] - nowadays, mobile devices such as laptops, cell phones, PDAs have the capability to access the Internet. Any user can start some work in the computer at the office and then continue the work using a PDA on the way back home for example. A mobile agent will do for good by keeping trace of its owner's tasks even while he is disconnected. Whenever the user goes online again, the agent will be transferred from its current location to the new one and the user can continue its work whatever the device he has selected to use. And best of all, the user will not notice how much the agent has eased his life.

- enhancing secure communications on public networks [18] - mobile agents transport certificates of the user they are working on behalf as they travel in the network. These are used for authentication at every node they stop by, therefore protecting them from eavesdropping. Mobile agents also travel along the network with data, code and process state fully encrypted to give a higher rate of security.

- Software distribution on demand [18] - Java applets and Active X are examples of implemented systems that widely use code on demand, creating indeed conditions

  to provide an alternative installation method: software distribution on demand. In fact, these systems are able to retrieve remote code and to install software packages automatically

without human interference. With all these features, they give emphasis to code mobility distribution and by consequence, to mobile agents.

2.1.4. Disadvantages of Software Mobility

The main difficulties of software mobility are associated to mobile agents' paradigm [5] [18]:

• Lack of applications [13] - currently, there are only a few applications using mobile agents. Even those, they cannot be considered successfully distributed among network users because they are not widely used. One of the reasons for this is perhaps that distributed systems are in general working satisfactorily well and not yet requiring the benefits that could be brought by mobile agents. Another reason could be related to the embryonic state of known experiences using mobile agents.

• Security concern [13] - besides the communication security improvements using software mobility, security has not yet been developed enough to create a comfortable environment to use mobile agents safely. This is an issue of particular sensitivity and unfortunately, the potential existing in the use of mobile agents can be done for good or for evil. Allowing remote software to execute locally without guaranteeing that it is not harmful is of critical concern. Many problems arise and research in this area has a long way to go.

2.1.5. Applications

Next, we state some application fields using mobile agents:

• Obtaining high quality information [5] - when a user seeks for some information in the Internet - using a search engine for example - the results found are more precise and more reliable to what the user was looking for because of the agent's ability, during the retrieval process, in collecting information from the sources that best match its owner profile.

• E-commerce [5,18] - the huge growth of electronic commerce verified in the last few years can certainly give mobile agents a more participative role in this field. As was stated, a mobile agent works on behalf of a user and it is equipped with intelligence to choose the most appropriate way that best serve the interest of its owner. If some user wants to buy, for instance, the cheapest airline ticket from Lisbon to Faro, the in charge agent travels along the network computers that provide this information to seek for the cheapest flight for this route. It may also use its owner Bank account to purchase the ticket for him or her. And in case there are no direct flights, the agent may even prepare a booking for a hotel room in some intermediate city for its owner to stay.

• Network devices supervision and configuration [5] - mobile agents can be used to supervise and configure network devices in the direction of giving the distributed system conditions to best perform continuously. Tracking down devices behavior is precious for

the development of upgraded components which will be sent through code mobility to the devices in need, increasing the network performance without human interference.

• Software maintenance and information collecting in LANs [5] - mobile agents can release users from irksome work such as installing and maintaining software in a distributed environment. Since a mobile agent is composed by an encapsulation of code, data and process state, it is able to carry instructions or software packages to install in remote computers without the user interference. A mobile agent can also be used to collect specific information that is spread along the network and which is in the interest of its owner. For instance, supposing that all medical records of a person, who went to various clinics and hospitals in his lifetime, are registered in local computers which are connected to the same LAN. Delegating to a mobile agent to collect all those sensitive medical records of a patient and making them available to the doctor when requested, releases the complex task which could have been assigned to a human being [11].

2.1.6. Execution Support

This subsection discusses how mobile software requires some mechanisms to successfully continue its execution in the destination computer. It comprises of state reconstruction and resource bindings, and communication.

2.1.6.1. State Reconstruction and Resource Bindings

In [5] Fugetta presents the state of a mobile agent is being constituted of:

• code - which indicates the static description for the behavior of a computation;

• data space - which contains the references to resources that can be accessed;

• agent state - which provides private data that is not sharable;

• execution state - which includes the run-time state of the process (program counter, call stack);

When an agent migrates to a remote computer environment, variables holding data, like open file descriptors, created when the agent was in execution in the source machine, may be void if they are directly accessed in the destination host. A structure containing a reconstruction of the resources bindings must exist to give support to the agent in successfully accessing them when it resumes execution in the new host. This structure is also important in supporting the agent to access resources that do not belong to the agent's address space and therefore, cannot be moved in the migration process. Nonetheless, when the resources are able to go altogether with the agent to the new host, there may be other agents requiring access to the same resources which have now become inaccessible. One solution to this problem is to create a copy of the resources in the computer where the agent is migrating - this process is called remote cloning - instead of

migrating the resources. Another solution for the resources that cannot be migrated is to access them as network references.
Also in [5] the composition of this structure is presented as Resource = <I,V,T> which:

- I is an unique network identifier for the resource;

- V is the value of the resource;

- T is the type of the resource;

A binding established by identifier - which is the strongest binding among the three - states that the execution unit must be always assigned to a given I (e.g. currently used printer). In regard of a binding established by value, V (e.g. desk jet printer 960dpi) must be associated, at any moment, with a given type and its value must remain unchanged when arriving to the destination host. T (e.g. desk jet printer) is the weakest binding and it is associated with a given type, independently what V and I are holding.

Furthermore, as mentioned above, not every resource can migrate. Resources mobility can be divided in three categories [5, 9]:

- free transferable - resources that are part of the agent's address space can move freely together with the agent to every network node;

- fixed transferable - resources that may be shared by other agents can only be transferred if they are not required by any other agent;

- fixed not transferable - resources that are physically attached to the computer environment cannot be transferred even if it isn't required by any agent;

In summary, the resources bindings' structure and the resources mobility attributes are mechanisms which contribute for the successful migration of mobile agents in distributed systems.

2.1.6.2. Communication

Agents may require communicating with other remote agents in some occasions. Although the concept behind mobile agents is to perform all the computations locally, in some occasions it is better to send messages remotely rather than migrating to the other agent's machine. The reason for this is that on some migrations, the amount of data transferred is much smaller than the agent's state. Therefore, remotely passing data messages instead of migrating the agent will benefit the efficiency and corporate work of the distributed system [18].

The reliability of remote messages passing is given by control mechanisms that work to guarantee that no message is lost. For example, in case a message does not reach the receiving agent, the sender must be acknowledged of this occurrence.

There are several types of communication:

- synchronous or asynchronous messages passing
- remote method invocation (RMI) or remote procedure calling (RPC)
- user-level communication protocols such as SMTP or HTTP
- distributed tuple spaces
- communication 1 to N

Due to mobile agent's ability to migrate autonomously to any network computer, it is difficult for other agents to know the exact location of an agent that they may want to contact at some point. Next, some strategies that aim to guarantee successfully deliveries of messages are described.

- Proxy [5] - the message is sent to the last known location of the agent. If the agent is not there, this host forwards the message to where the agent has gone next and so on, until it reaches the agent. In extreme situations, the message behavior will look like it is chasing the agent;

- Dynamic snapshot delivery [5] - each network node holds a copy of the message until it is delivered to the agent;

- Based on a distributed system [18] - the system's naming service (which may be distributed) give support to every mobile agent to be referenced and therefore, reachable. In this method, no longer is important where the agent is but the name that it holds. Thus, every mobile must have a name and must inform the names service whenever it is going to migrate. On the occurrence of a message has not been received by its recipient agent (A), the sender (B) will request the location of A to the names service which will be provided to B. Then, B will try again to contact A.

2.1.7. Systems Supporting Software Mobility

The Java language has captured most of the attention on software mobility technology because it is able to provide a platform-independent language among agent applications by using the JVM. Unfortunately, due to Java imposed restrictions related to security measures, JVM supports only the migration of code and data (weak mobility). However, if the JVM is subject to modifications, it enables Java systems to support strong mobility. Sumatra [2] is an example

of a language for resource aware mobile programs which features strong mobility by modifying the JVM. There are also implementations of strong mobility for multi-threaded agents in Java [32] in which each agent thread maintains its own serializable execution state at all times, while thread states are captured just before a move.

There are systems that transform programs that use strong mobility into programs that rely only on weak mobility [30]. KLAVA [31] is an experimental Java package for distributed applications and code mobility that offers this kind of support.

Aglets [8], Voyager [6], Gypsy [7] and a few more, are examples of systems that also use Java packages to implement systems supporting mobile agents.

But there are also systems that support mobile agents that are not implemented in Java. Of these, we highlight Telescript and Mob.

Telescript [27] is one of the first languages to use mobility on the development of loosely coupled distributed applications whereas Mob [21] is, to the best of our knowledge, the first language to combine services and mobile agents. Mob is a mobile agent scripting language where agents implement and require services, thus providing agent anonymity in inter-agent communication. There is no notion of session uploading as agents access services provided by other agents through remote method invocation. Mobility in Mob is related to the fact that the whole computation (the agent) has the ability to move. Indeed, this mobility is done towards hosts and not service-oriented. Additionally, Mob also supports strong mobility by running on a dedicated virtual machine on top of the JVM.

In this section, we have discussed software mobility models, advantages and disadvantages of software mobility, how a mobile agent is transferred from one host to another, applications and systems supporting software mobility. In the next section, we will describe the Web services paradigm.

2.2. Service-oriented paradigm

The lecture of this section is centered on the service-oriented paradigm. We start by introducing the principal distributed objects architectures that preceded service-oriented architectures. Next, we will discuss service-oriented architectures which are the platforms that support Web services. The objective of this section is to complement the Software Mobility paradigm described in the previous section in order to set the bases for Service-oriented Mobility, the theme explained in Chapter 3.

2.2.1. Distributed Objects Architecture

2.2.1.1. CORBA

The Common Object Requesting Broker Architecture (CORBA) [12, 16] was one of the first infrastructures to appear which looked for tackling down distributed systems complexity. CORBA strategy focuses on promoting interoperability and adaptability between network components created by different computer languages. It basically uses Object Request Brokers (ORB) to provide an understanding platform between components written in different programming languages and an Interface Definition Language (IDL) which is a neutral-language presentation of each component in the referred platform, so others components can request to operate together. In addition, CORBA technology can work together with the Java platform to enhance its portability and productivity.

2.2.1.2. Java RMI

Java Remote Method Invocation (Java RMI) [23] is a technology based in Java which has the particularity of being able to invoke methods on Java objects which are remotely located and make use of these objects as if they were locally present in the invoking computer. It is typically constituted by a RMI server which is responsible for creating the referred Java objects and a RMI simple naming facility - RMI registry - which is a storehouse of references to the objects created by the RMI server. RMI also provides a mechanism through which is performed the communication between network computers running RMI applications (e.g. a JVM program running on the client machine invokes a method on an object located in the RMI server).

What Java RMI introduces successfully is the ability of a client JVM not requiring creating locally an object which is remotely distributed to perform any task on it. The client JVM application invokes methods on the remote object and the results or effects produced is what is sent back to the client. These methods can be either provided at the server side or at the client side but it is required that they implement some particular interface. In addition, a single distributed object can be used concurrently by any number of client applications because Java RMI creates a local stub (that is basically the remote reference) in the client JVM which acts as a local representative of the remote object.

Another feature of using Java RMI is that for the software engineer, invoking remotely located objects looks similar to regular Java method invocation.

2.2.1.3. DCOM

Distributed Component Object Model or DCOM [33] is a proprietary Microsoft technology for building distributed software components in networked computers. DCOM is an extension of COM (also from Microsoft) that supports communication between objects in distributed

environments and features the development of applications that focuses on centralizing business rules and processes, provides scalability and facilitates maintenance. Additionally, DCOM works transparently for both the client and the server application which are encoded according to the COM standard.

2.2.1.4. EJB

Enterprise JavaBeans or EJB [34,35] is an embracing technology that offers an infrastructure for constructing corporate server-side distributed Java components. It offers an architecture that supports distributing components that integrates several requirements at the corporate level, such as distribution, operations, security, transactions, persistence and connectivity with mainframes and Enterprise Resource Planning. In comparison with other technologies consisting of distributed components (CORBA and Java RMI for instance), the EJB architecture occults the subjacent system-level semantics that are used on distributed component applications.

2.2.2. Service-Oriented Architecture

Service-oriented architecture (SOA) [10, 17] is a kind of computer network architecture which basically lets applications and systems make use of services available in a distributed system. No matter what size is the distributed environment (e.g. World Wide Web), this architecture facilitates the integration of any applications and systems, and also reduces the effort expended by software engineers in developing new ones. This is possible because in SOA, applications are in the form of services providing therefore universal interoperability between them. Thus, functionalities or services which are currently being offered in the network can be used by other services as well as the new ones that will be implemented. This feature allows software engineers to develop new services which could make use of existing ones. With this strategy, a service – which may request to use other services – can provide any functionality without much hindrance for the developer, from the simple ones (e.g. give the currency exchange value of 100 USD in Euros) to the more complex ones (e.g. booking the cheapest flight from Lisbon to Faro with one night stay in some hotel).

The main feature introduced by SOA is allowing client applications (or services) to make use of multiple services no matter what code language or platform is used by the service. This is called loose-coupling. A client application (or service) does not need to have much knowledge about the service in order to use it. It just needs to know that it exists and the operations it offers, so it can be requested when needed. In addition, the software engineer that is developing the client application does not have to worry about how it communicates. What must exist is a well-defined interface for each service to be accessible by other clients or services in order to communicate with it. In case a revision has to be done on an application (without changing the service offered), it is the application that has to be changed and not the interface. This is a very helpful solution because applications can be updated without changing the service offered and therefore, does not need to inform other clients regarding of the application revision.

The mechanism involving SOA services is typically supported by a find-bind-execute paradigm (Fig. 2.3). In this paradigm we have a service provider, a service registry and a service consumer. The service provider creates services which are registered in the service registry which in turn is a public directory of services. Whenever a client requires a specific service, it looks at the service registry and if there is any matching, a contract and an endpoint address for that service is given to the consumer by the service registry in order to be able to request it. The 'new' terms contract and endpoint address are in fact, constituents of the previously mentioned service interface.

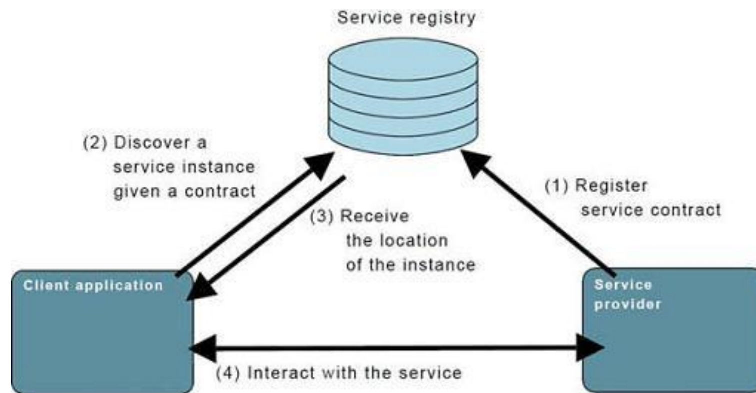


Figure 2.3 - Find-bind-execute paradigm

2.2.3. Web Service Protocols and Technologies

The Web Services approach is based on a maturing set of standards that are widely accepted and used [17]. It provides a common understanding platform between clients and services which traditionally communicates through applications developed in different computer languages. The Web Services standards are based on Web technologies XML HTTP namely SOAP, WSDL, and UDDI which are described below. Since security and asynchrony are important in the context of our work, we also discuss the standards WS-Security and WS-Notification.

2.2.3.1. Communication

Although XML along with its schema provides a common computer language for any computer environment to understand the content existing in a Webpage, it is necessary to use an agreed- upon format for the communication process. SOAP means Simple Object Access Protocol [3] and it is an XML-based protocol for exchanging information between clients and services in a network. A SOAP message consists of an envelope, a header which is optional, and a body. Through the envelope, an XML namespace and an encoding style are sent. It is vital to specify the names in the message to avoid ambiguities between names attributed to different items. The encoding style is used for identification of data types. The header which is optional is used to provide additional information for an intermediate node to deal with the

message in case it has received. This information is often related to security issues. The body which contains the essential part of the message is for the destination node.

2.2.3.2. Service Description

XML and SOAP solely are ineffective if the client does not know how to access to Web services and which operations they perform. A typical Web service may not have one interface only and interfaces may not provide just one operation. It is important for a Web service to be presentable to the clients in order to be easily requested. The way a Web service 'becomes' organized with all its operations, interfaces and protocols are provided by a WSDL document. WSDL or Web Services Description Language [4] is an XML format document with definitions describing the operationality of a Web service. A WSDL document makes a service viewed as a collection of network endpoints or ports. These ports relies on URIs (Uniform Resource Identifier) which the service uses to communicate (send or receive messages) with the exterior through HTTP, SMTP, TCP, etc. Each port is unique and it is created to support specifically a set of operations of the same type. In summary, WSDL documents group messages into operations and operations into interfaces and these in turn are bonded to the ports or endpoints (Fig. 2.4). By presenting an understandable format of what a Web service can offer, WSDL document provides to any client how to appropriately make use of a Web service.

2.2.3.3. Service Registry

Universal Description Discovery and Integration (UDDI) [1] is a standard used to describe Web services registered in the registry repository which aims to provide clients a method to easily find a desired service. A UDDI registry has 3 main components which give a better organization of its content and enhances thus the clients searching: yellow pages, white pages and green pages. White pages typically hold information regarding of the business providing the service, such as its name, its description which could be in one or more languages and contact information (phone number, email, address, etc). Yellow pages can be viewed as a sub-group of the white pages as they are grouped by services of the same business name. A business company can have one yellow page only if it offers a single service. Each tuple also holds a description of the service. And finally, the green pages typically have information regarding of how to access a service's interface binding (a service could have one or more bindings). A service could have one or more green pages because services could have one or more bindings.

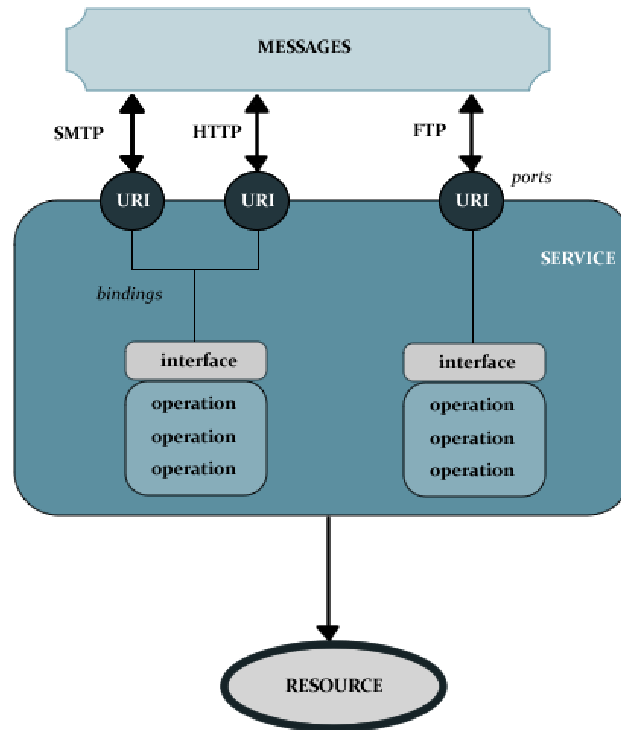


Figure 2.4 - Organization of a Web service through WSDL format

2.2.3.4. Security

XML, SOAP, WSDL and UDDI provide the conditions for a client to find a needed service and understand how to make use of it, independently of where the client and the service are located in the network. However, these standards are not enough to be applied in a Web services-based SOA. Security is also an important issue for the sustainability of distributed systems with Web services.

WS-Security [15] is the protocol currently in vigor (released by OASIS in February 2006) and it relies on security tokens to provide integrity, confidentiality and authentication in a SOAP message. For instance, the authentication process can be performed by combining the security token with the sender's digital signature. Thus, the receiver is able to acknowledge the veracity about the author of the message.

When integrity can be guaranteed in a SOAP message, it means that the message will not suffer any changes during its traveling. When confidentiality is assured, it is synonym that the message will not be read by any party except the intended one. And when authentication is made, it helps to prove that the originator of the SOAP message is correct. All these mechanisms along with cryptographic technologies can be combined together in several ways to provide various security models for the distributed environment. WS-Security is also extensible to other mechanisms which aim to enhance security in SOAP messages exchanging in a Web services- based network. In addition, WS-Security also describes how to build binary security tokens encryptions that can be included in SOAP messages.

2.2.3.5. Web Services in Java

There are frameworks that give support to applications which would like to interact with Web services. As mentioned before, SOAP is the protocol which Web services use to communicate between them and in which SOAP messages go through. So, what these frameworks typically do is to build and interpret messages, acting thus as a SOAP engine between applications and Web services.

AXIS 2 [25] is an example of these frameworks and it is implemented in Java. Although Web services use XML language, AXIS 2 provides an environment in which is possible to create Java applications that can ’directly’ communicate with Web services. Another popular framework is the JAX-WS [24] which is a Java programming language API. Briefly describing, JAX-WS provides an environment that simplifies the task of software engineers in developing Web applications and Web services.

3. Service-Oriented Software Mobility

In this chapter, we will present the combination of the software mobility and the service-oriented computing paradigms explained in Chapter 2.

3.1. Overview

Software mobility [20] is a well-studied and known paradigm for the programming of distributed applications. The advantages offered by this technology such as abstracting the underlying network and reducing the requirement to maintain costly network sessions have been off-staged by well-founded security concerns [28, 29]. In spite of the security reasons that have restricted the use of the paradigm, we have the conviction that software mobility can still be a useful technology in trusted environments such as local area networks and middleware layers that assure control of mobile agents.

Service-oriented computing has emerged as a technology that provides abstractions of both network resources and software components. Indeed, the modeling of distributed applications into loosely-bound service-oriented components has been proved to be a good paradigm, especially in heterogeneous environments, such as the ones used in mobile and grid computing.

Service-oriented Software Mobility [19] is a paradigm that combines software mobility and service-oriented paradigms. This technology consists of supporting the migration of components regardless of their origin or developer throughout a network that is presented in the form of services instead of network nodes. In other words, this environment allows user-developed components to migrate towards a service provider instead of the classic migration towards a machine. The objective of this thesis is to instantiate a model that uses this technology as proposed by Paulino.

It is important to mention that our goal is not to replace the usual client/server interaction technologies but rather to provide a simple and transparent way for programmers to make use of the known benefits of software mobility.

In service-oriented computing, services are presented in a transparent way for clients, requiring only knowledge of a contract which is the service interface. This feature offers anonymity and the kind of loose bindings desirable to construct resilient programs in highly dynamic networks, such as the ones consisting of mobile devices. As such, an application is modeled in

terms of distributed inter-connected components which communicate based on the client-server paradigm.

Typically, the operational flow of a program that wants to perform the sequence of operations P1 to Pn at hosts h1 to hn is:

go to host h1 and do P1;
go to host h2 and do P2;
...
go to host hn and do Pn;

But in an environment which is service-oriented, the operational flow of a mobile agent is:

find an instance of service S1 and do P1;
find an instance of service S2 and do P2;
…
find an instance of service Sn and do Pn;

To find an instance of a service, a session requires obtaining the location of the service provider before departing from the client machine. A services repository is used in such service-oriented distributed environment with the objective of registering services available in the network. When a new service is 'launched', the service's contract is stored in this repository along with the location of its provider. Thus, before the application leaves the client machine, it needs to contact the service repository to obtain the location of the host providing the service it is looking for (Fig.3.1.1). It is important to realize that this process is completely abstracted to the programmer as the requisite to migrate is to specify the service that the session is defined to look for and not the host.

This process is completely hidden to the programmer thanks to a middleware layer that is responsible for getting the location of the service provider by querying the service registry. The session is then uploaded to the host providing the service and when the interaction is over, the computed result is returned to the middleware which sends to the client application.

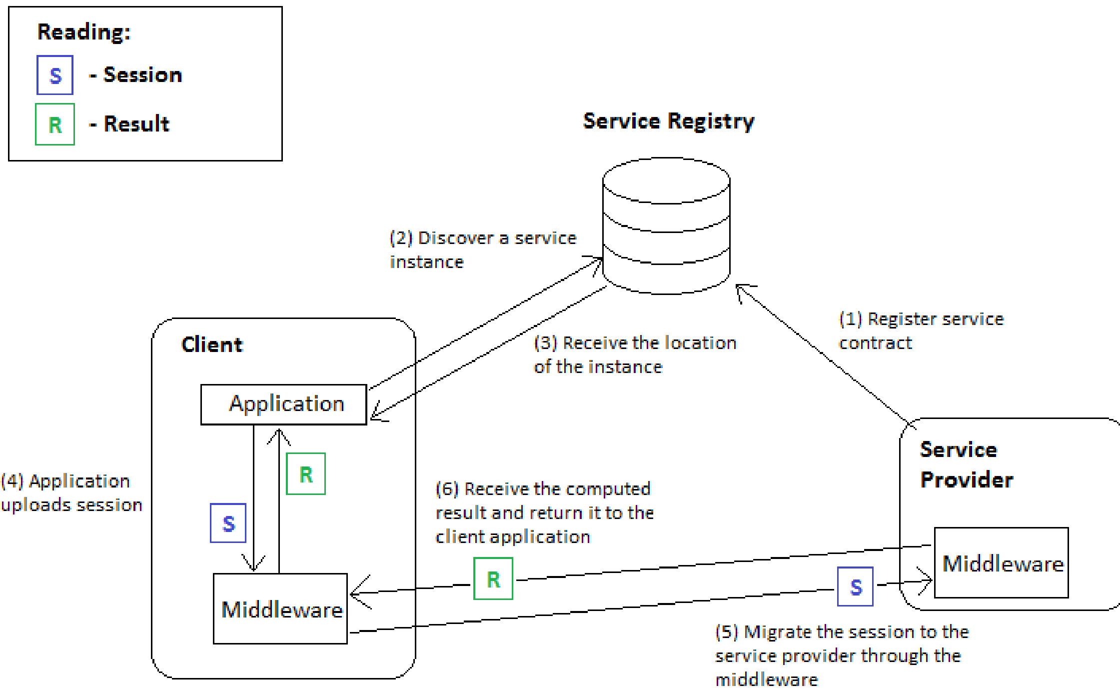


Figure 3.1.1 – Session uploading in a service-oriented system

One of the benefits brought by the inclusion of service-orientation is that it removes the burden that the programmer has in implementing the code to overcome the possible absence of a host (and when this happens, to search for alternatives). Since mobility is modeled in terms of services, rather than hosts, the handling of alternatives is responsibility of the service discovery mechanism.

Another advantage of using services in software mobility technology is the enhancement of security. Access to local resources can be encapsulated into services, which obligates all host machine interaction to use service providers as intermediates. Proof-carrying code can validate that an uploaded session does not try to access the local resources directly.

3.2. Itinerary

A session can have its mobility enhanced if it is defined to migrate to a set of hosts. This means that a session is able to travel further than a single service provider (Fig.3.2.1).

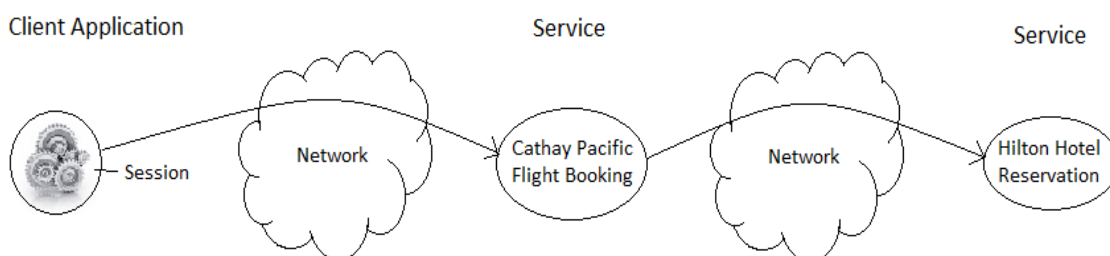


Figure 3.2.1 – Session migrating to two services' providers

Figure 3.2.1 represents a session migrating to the service provider of “Cathay Pacific flight booking” and then, to the service provider of “Hilton Hotel Reservation”. The itinerary feature is very useful for sessions that have a set of tasks to be performed remotely. In a trusted environment, a session can be programmed to interact with N hosts and to successfully accomplish the tasks that have been defined to do without returning to the client. This means that the client dispatches the session only once regardless of the number of servers that it will interact with.

The illustration described above, contains a sequence of events for a session featuring itinerary. The session is programmed to make a flight booking in Cathay Pacific airline and to make a reservation in Hilton Hotel. Indeed, the programmer is not aware of the location of the servers that provide these services at any time. It is the service’s instance that indicates the location of the host that provides the service which the session is looking for. The programmer is only required to specify the services and to upload the session. Once uploaded, the session becomes under the responsibility of the system that is defined to take it to the hosts that offer the requested services. A solution to overcome this problem is to have a middleware layer in the client machine that is in charge of finding an instance of the location that provides the requested service.

No matter how many hosts a session visits, it has contact with the client machine in two occasions only: before being uploaded to the network and once the work is completed. Indeed, two of the premises of the mobile agent paradigm are: enabling disconnected execution (the client does not have to be connected to the network to provide session execution) and autonomy (the session must be able to make choices without the user’s intervention).

The time that a session takes to complete its tasks, i.e., to return the results to the client, may vary immensely. Thus, it is our opinion that asynchronous communication is more adequate for environments that manage user-developed components.

3.3. Unidirectional traveling

The itinerary feature allows us to perceive that a session can be defined to obey to a specific traveling strategy. Unidirectional traveling is a strategy in which sessions travel from host to host in sequence (Fig. 3.3.1).

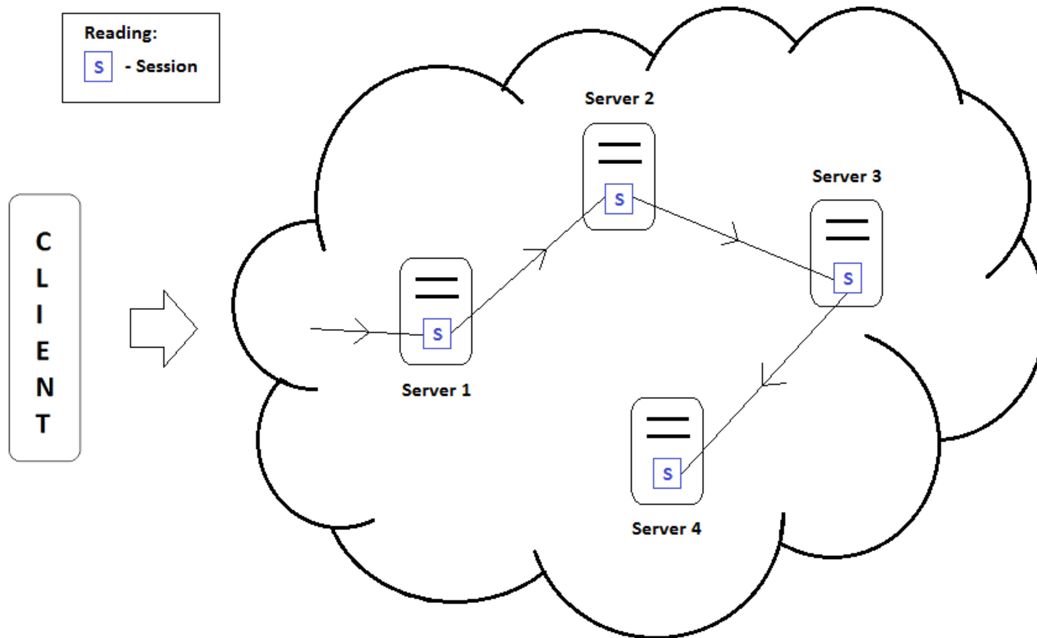


Figure 3.3.1 – Upload with unidirectional traveling strategy applied

An evident advantage in this strategy is when the tasks defined in the session have dependencies among them, i.e., a task to be executed in server 2 can only be performed after the session has executed in server 1. An example of this approach is represented in Figure 3.2.1. Supposing that the user intends to travel to Hong Kong flying Cathay Pacific and to stay overnight in a hotel (the user loves the chain Hilton). The unidirectional strategy 'obligates' the session to make the reservation of the room only after the flight booking has been confirmed. Thus, making a room reservation before the flight being booked is an unwelcome situation that never happens.

In this strategy, the contact with the source machine occurs only when the computation on the last server is finished. When the results are returned to the client, it means that the session has completed its job.

3.4. Multidireccional traveling

A second traveling strategy that can be featured in a session regards to a parallel migration. In this strategy, the session is defined to migrate to a set of servers at the same time (Fig. 3.4.1).

Unlike the unidirectional strategy, the source host establishes contact 'at the same time' with all selected nodes that the migrating session is indicated to work on. This strategy is adequate for a session that is defined to do the same job in a set of hosts. For instance, supposing that the task is to install a software package in all machines in an office. In this procedure, prior to the uploading, a copy of the session is required for each of the servers specified to migrate to.

Since the execution of this job is done in parallel, the time expended to accomplish this task in all servers is lower than if done in sequence.

And when the execution is completed in a given host, the session will not migrate to another node rather than returning to the source. Since this is a service-oriented environment, the client application doesn't have direct contact with any host. Thus, the system must have mechanisms to wait asynchronously for all results to return to the client machine. When the results have arrived, the system delivers them to the client application.

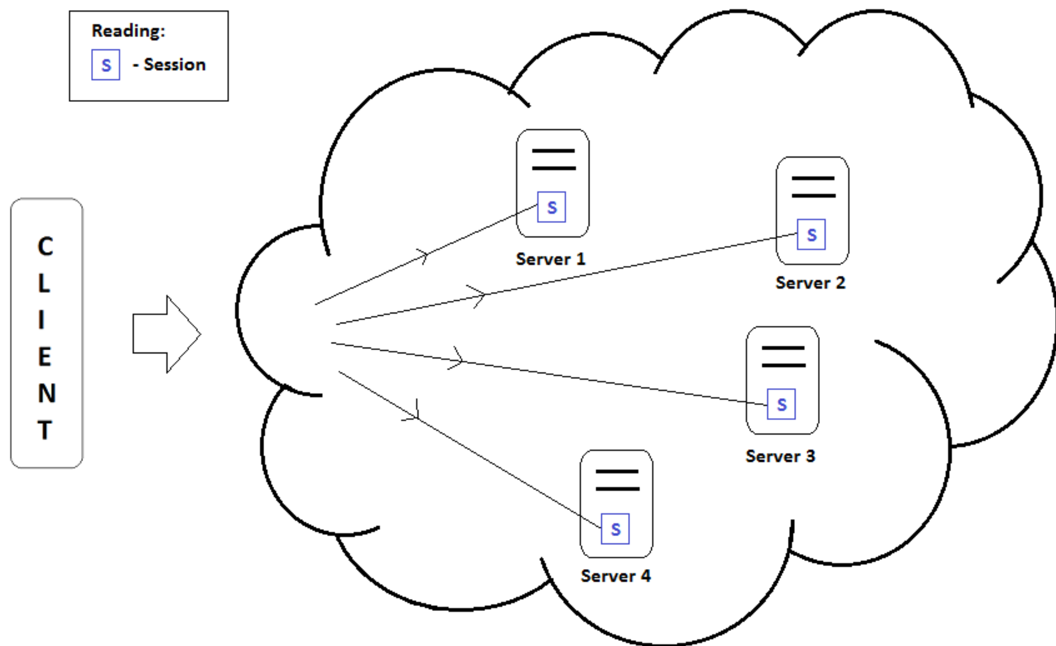


Figure 3.4.1 – Upload with multidirectional traveling strategy applied

3.5. Bridging

A feature we found interesting to include in our model is related to the bridging of sessions that have 'common interests'. This characteristic refers to the situation in which a session uses the computed results of another session as part of its execution.

This feature is only possible in a system that accepts a set of sessions to be uploaded together at the same time. The first session is executed in the servers specified to do so, and then the second session migrates to the servers defined to work at, and so on (in case there are more sessions). It is important to mention that the sessions require traveling together for the computed results of one session to be copied to the other when the former completes execution.

In fact, there are two ways to copy the results from one session to another: one is doing it manually (Fig. 3.5.1), i.e., the programmer waits for the results of the first session to return and

then adds it to the second. In case the first session takes a long time to deliver the results, the second one is 'never' uploaded. The second solution (Fig. 3.5.2) is to allow sessions to interact with each other after being uploaded. In other words, when a session completes execution in a given host, in case it has a bridge defined to other session, the computed results are copied to the latter. This process requires that both sessions are sent together as the results' copying is done automatically by the system and distant from the client.

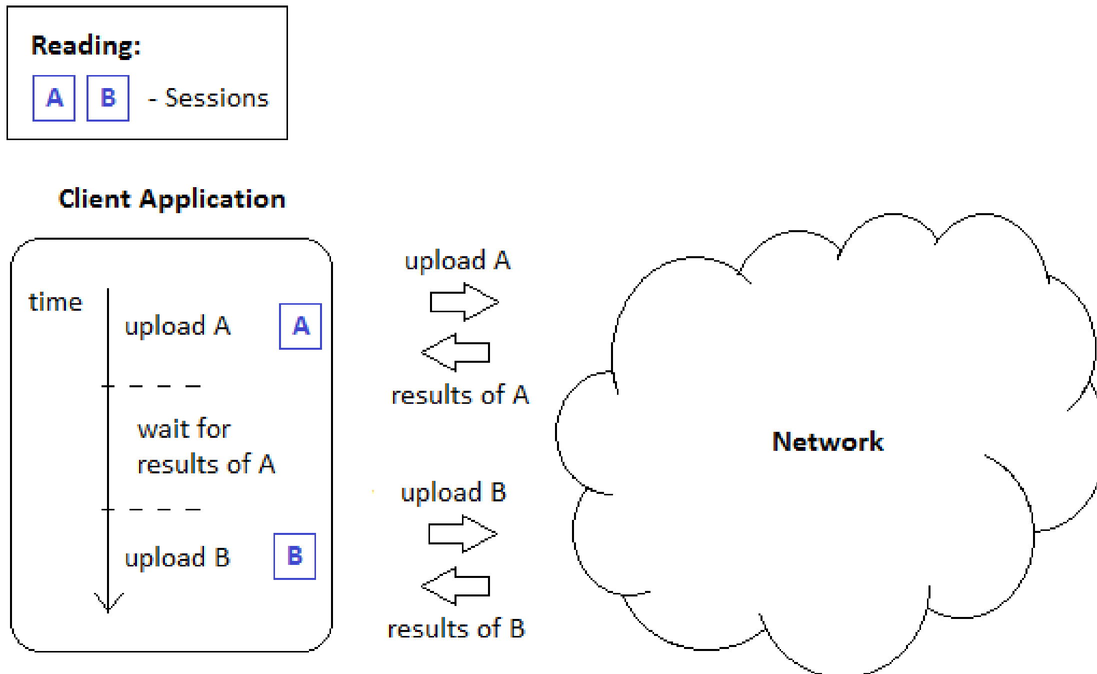


Figure 3.5.1 – Manual bridging

In the scenario represented in Figure 3.5.1, the results copying is done in the client, after the results of A have returned. Additionally, session B can only be uploaded after the results of A have been copied.

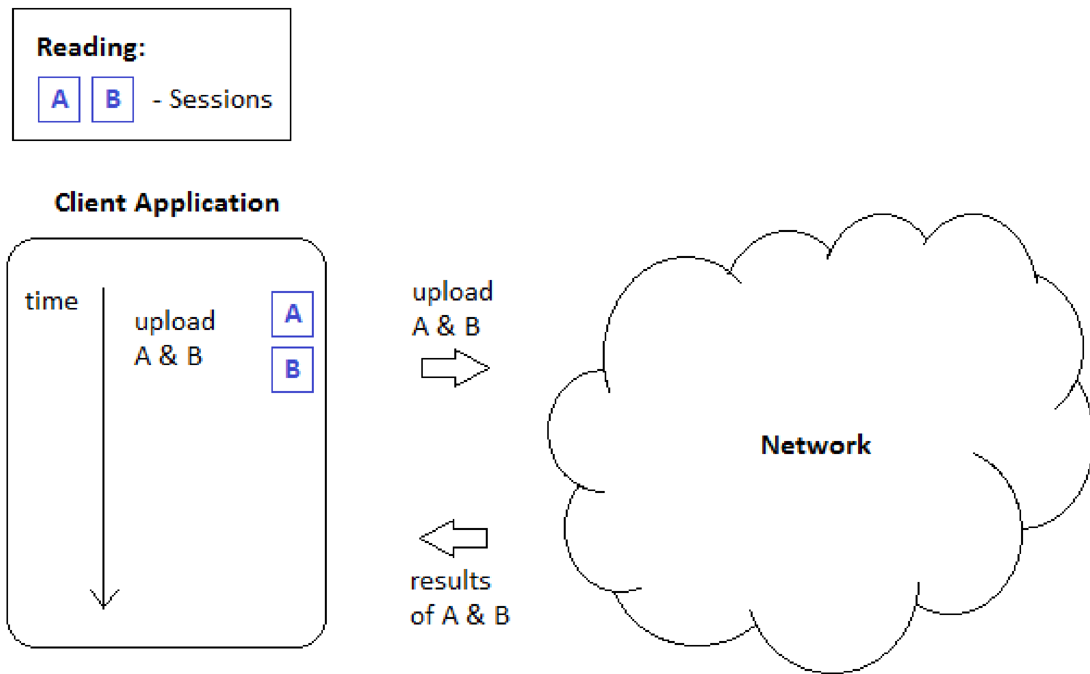


Figure 3.5.2 – System bridging

In the scenario illustrated in Figure 3.5.2, the situation is completely different. Session A and session B are uploaded together to the network. When the execution of session A is completed, the respective results are copied to session B. This process is done remotely from the client, i.e., in some host in the network. Then, when session B finishes its computation the results of both sessions are delivered to the client.

An evident advantage provided by this feature is that the client only needs to define the sessions that make a bridge prior to their uploading as the results copying is done by the system in the network. Therefore, the client is not required to wait for the results of middle computations to return in order to use them in another session.

4. Instantiation of the Service-Oriented Mobility Model

This thesis focuses on an instantiation of the service-oriented mobility model in the Java language with the use of Web service technology for communication between remote hosts.

In this chapter, we start by providing an overview of the model instantiated (Section 4.1). Then (Section 4.2), we present the application programming interface (API) that offers a general guidance for the user to understand how to prepare a session to be uploaded to the distributed system in Section 4.2. In Section 4.3, we introduce the architecture of the system. In Section 4.4 we present the lifecycle of an itinerary. Finally, in Section 4.5, we provide an in-depth description of the implementation of the system middleware.

4.1. Overview

The objective of this work is to instantiate the service-oriented mobility model so that the sessions programmed in Java language can be locally attended by services – preferentially Web services – available in remote stations.

In Figure 4.1.1, a general architecture of the system is illustrated and it consists of a client and several servers running a middleware layer. User-developed components – sessions or itineraries – can be submitted by the client to the network through methods available in the proposed API (see Section 4.2).

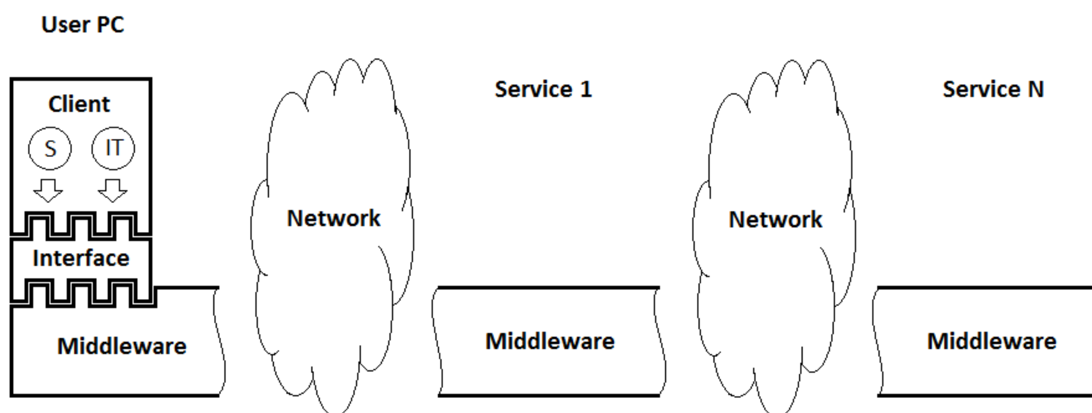


Figure 4.1.1 – General overview of the system

The middleware is a software layer distributed among all machines willing to receive and execute sessions. This means that there will be one process running in each machine listening to receive and execute user-programmed components. Thus, it is of the responsibility of the middleware to transport sessions – and itineraries – to the server which offers the requested service. In other words, the middleware has to take care of the user-programmed components from the instant they leave the client until they have completed their trip in the network. Once the trip is completed, the remotely computed results must be returned to the client. This is the purpose of this work.

One of the advantages brought by the inclusion of a middleware – along with the API – is that it has removed the burden a user has in the process of uploading a session. The middleware assures the responsibility to create the conditions required for a user-programmed component to be well succeeded when executed in a remote host. For instance, a session may require Java classes that do not exist remotely. It is the job of the middleware to find these classes and pack them with the session that was requested to be uploaded. In Section 4.3, we will have a more in-depth comprehension of this middleware layer.

4.1.1. Scenarios

In order to have a better understanding, Figures 4.1.2., 4.1.3 and 4.1.4 demonstrate three different scenarios possible in the instantiated model (disregarding the traveling strategy applied): one-to-one, one-to-many and many-to-many.

The *Home* reference (represented with an "H") is used for the itinerary to have knowledge of the "home address" of the client. It is utilized to deliver the computed results.

One-to-one scenario happens when the client uploads one session that migrate to one server only (Fig. 4.1.2). Thus, a single result is returned to the client once the trip is completed.

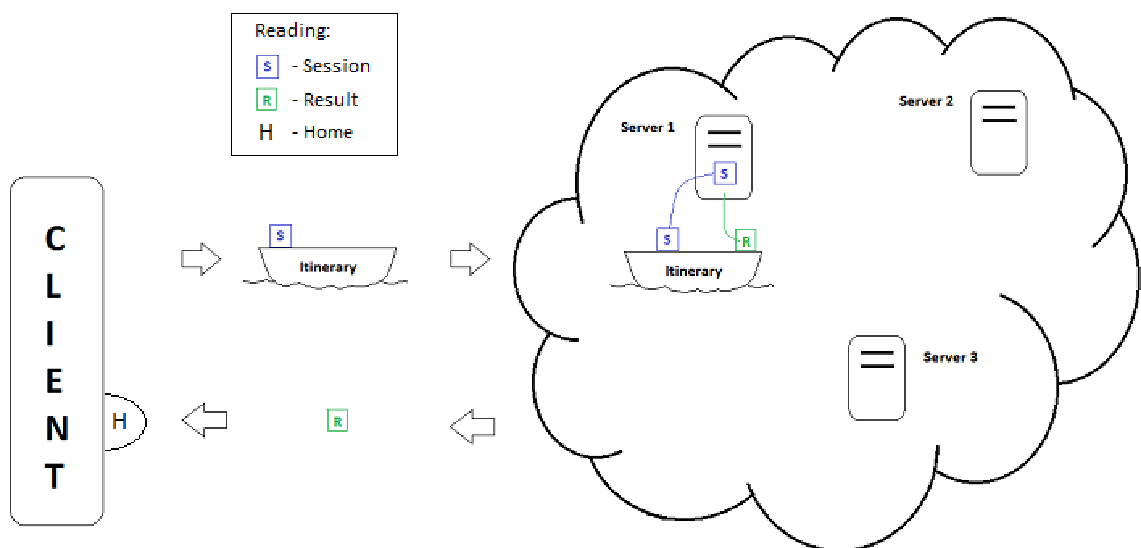


Figure 4.1.2 – One-to-one scenario

One-to-many scenario happens when the client uploads one session that migrates to many servers (Fig. 4.1.3). Thus, many results are returned to the client once the trip is completed.

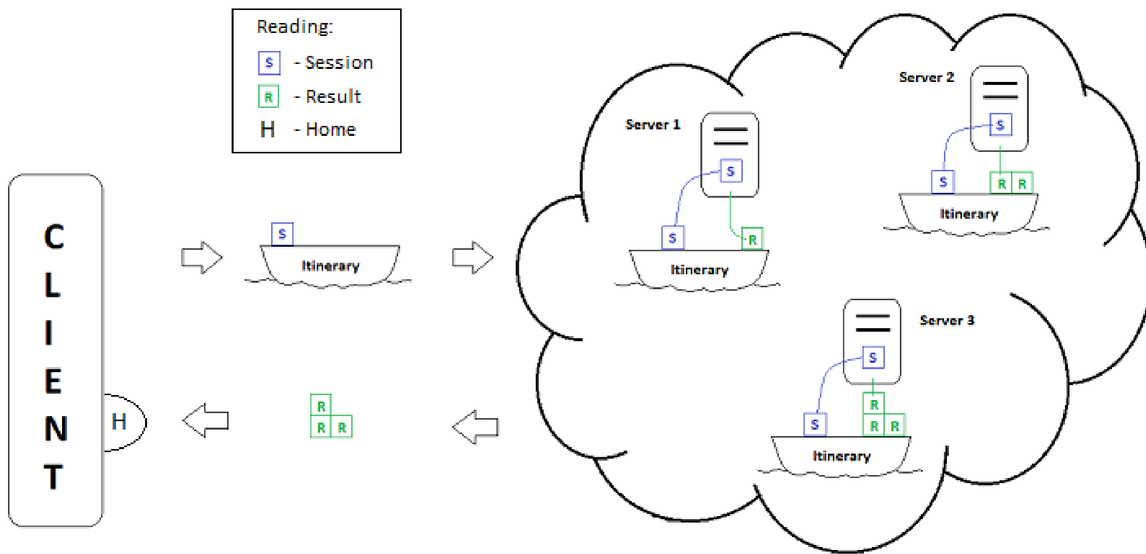


Figure 4.1.3 – One-to-many scenario

Many-to-many scenario happens when the client uploads many sessions that migrate to many servers (Fig. 4.1.4). Thus, many results are returned to the client once the trip is completed.

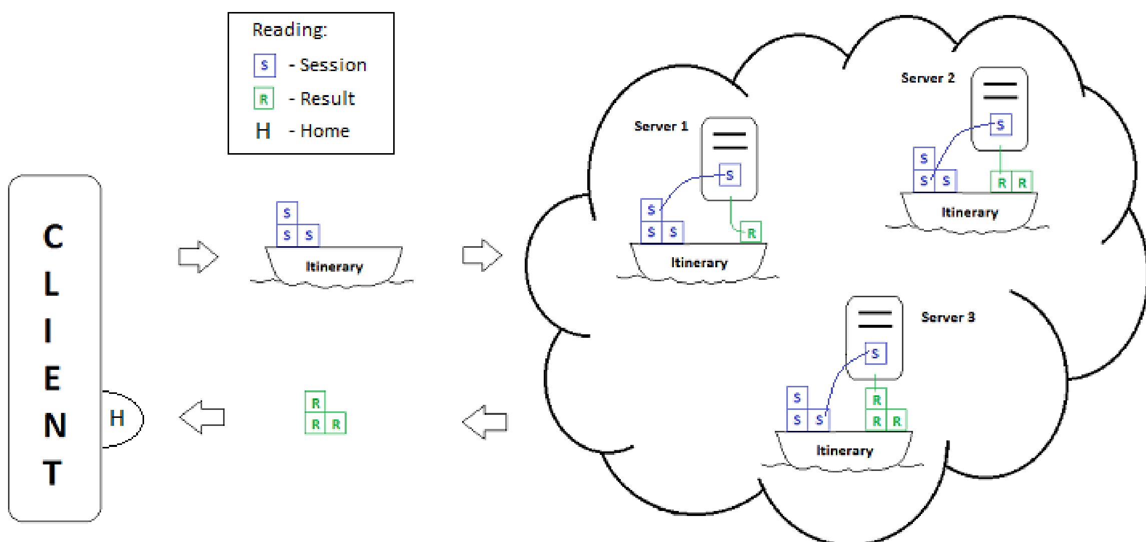


Figure 4.1.4 – Many-to-many scenario

4.1.2. Data Manager

A feature we found interesting to include is a data storage component which we called data manager. In this model, the client has total freedom to write the code which is going to be

executed remotely. Thus, by providing a data manager, the client is able to implement a session that stores and retrieves data to and from a storage recipient.

A data manager is featured to be local to the session or remote. In other words, this means that a session is able to interact with a data manager that travels together with the session or a data manager that is available on a remote computer. The benefit of using a remote storage facility is that it reduces the content laid up in the session during its trip, especially when it has stored large amounts of data.

Next, we will present the user application programming interface which offers mechanisms for the user to develop sessions and to upload them to the network.

4.2. User Application Programming Interface

The platform provides an API to implement sessions which consists of Java classes offering methods that interact with the middleware – e.g. upload of a session.

One of the benefits coming from the use of middleware is that it allows multiple processes running on one or more machines to interact. Thanks to the middleware, the user workload is reduced to the essential (session implementation) as the middleware disposes of mechanisms associated to the management of user-programmed components dispatched to the network. For now, we will present the classes that make up the API component and justify how to use it, leaving the middleware implementation to be explained in Section 4.3.

4.2.1. Server

In the original model, the migration's target is a service, i.e., the migration is done towards the location of the service to which the session wants to interact with. The abstraction featured in this instantiation, a server provides a more general concept. In other words, a server denotes the identification of the location where a session can be migrated to. For example, the URL address of a server, a service's name for books purchasing (e.g. Amazon), a service's name for flights booking in a travel's agency and so on. In fact, a Server can even represent a repository (such as an UDDI directory) upon which the location of the target service can be queried.

Listing 4.2.1: The `Server` interface

```
public interface Server {
  URL get();                  // returns an URL of this Server
  List<URL> getAll();        // returns a list of URL of this Server
}
```

Interface `Server` contains two methods which return a URL and a list of URL as it is illustrated (Listing 4.2.1). Thus, any session could be uploaded to any kind of server that implements `Server`.

4.2.2. Result

The result of executing a session is abstracted in class `Result` (Listing 4.2.2).

Listing 4.2.2: The `Result` class

```
public class Result<T> {
  public String getOwner();   // returns the owner of this Result
  public T getResult();       // returns the result computed
}
```

This class enables a user to acknowledge the computed result through method `getResult()` and the name of the machine where the result was computed through method `getOwner()`.

4.2.3. Session

`Session` (Listing 4.2.4) is an abstract class that implements interface `SessionRunnable` (Listing 4.2.3) which provides a method called `run()` for the programmer to specify the code to be executed remotely.

Along with class `Itinerary` (Listing 4.2.7), class `Session` provides the methods for the user to develop components to be uploaded to the network. To create a session, the user is only required to develop a class that extends class `Session`.

Listing 4.2.3: The `SessionRunnable` interface

```
public interface SessionRunnable<T> {
  T run();
}
```

Listing 4.2.4: The `Session` abstract class

```
public abstract class Session<T> implements SessionRunnable<T> {

  // Uploads this session to the location identified by server and
  // returns a Future Result
  public Future<Result<T>> upload(Server server) {…}

  // Uploads this session to the location identified by server and
  // returns a Future List of Result
  public Future<List<Result<?>>> uploadAll(Server server) throws
         HomeCreationFailedException {…}

  // Uploads this session to the locations identified by servers and
  // returns a Future List of Result
  public Future<List<Result<?>>> uploadAll(List<Server> servers) throws
         HomeCreationFailedException {…}

  // Executes this session locally and returns a Result of T
  public Result<T> exec() {…}
```

```
    // adds a data manager identified by the given key
    protected void addDatamanager(String key, DataManager<?,?> dm) throws
           KeyNotAcceptedException {…}

    // Returns the data manager associated to the given key
    protected DataManager<?,?> getDataManager(String key) throws
           KeyNotFoundException {…}
}
```

Class `Session` consists of three different methods to upload a session to the network and one method to execute it locally. All upload methods require the identification of the location where a session could be migrated to.

Method `upload` uploads the representing session to a single location identified by an instance of `Server`. In other words, when this method is invoked, the session is uploaded to a location given by `server.get()`.

Method `uploadAll` uploads the representing session to all the locations identified by an instance of `Server`. It has two versions: one that receives the list of the locations to travel to, and the other that retrieves these locations from a single server parameter by invoking `server.getAll()`.

It is important to take into account that both methods `uploadAll` may throw a `HomeCreationFailedException`. This situation happens when the system was not able to create the mechanisms to return the computed results to the client. This `Exception` is not thrown by method `upload` because the session is uploaded to one location only, which means the connection established between the client and the server can be used to return the computed result.

In regard of the values returned by each of these methods, all of them return instances of class `Future`. Class `Future` belongs to the Java concurrent library (`java.util.concurrent`) and abstracts the result of an asynchronous computation. The reason behind the use of a future is that the time expended for an upload to return its results could vary immensely. By attributing a future to the returning result, the thread responsible for this computation waits passively for its results to return, benefiting thus the processing on the client computer.

Method `exec()` simply executes the session in the local machine. This feature is useful on the occasion that it is not possible to upload a session to a remote host because the target machine does not feature our middleware or simply does not allow this session uploading. This method executes the session in the local computer and interacts with the network resources remotely.

Regarding of methods related with data managers, `addDataManager` adds a data manager associated to the given key and a `KeyNotAcceptedException` is thrown if there is already one stored with the same key. Method `getDataManager` returns the data manager associated

to the given key and a `KeyNotFoundException` is thrown if there is no data manager stored with this key.

4.2.4. Data Manager

Abstract class `DataManager` (Listing 4.2.5) was developed with the objective of being used as a data storage recipient while a session is running. Thus, it is up to the user to define when the session interacts with the data manager.

Listing 4.2.5: The `DataManager` abstract class

```
public abstract class DataManager<K,V> implements Serializable {


  // Constructs a DataManager with the String dm_key
  public DataManager(String dm_key) {…}

  // Returns the key of this DataManager
  public String getDataManagerKey() {…}

  // Returns the value V referenced by K key
  public abstract V get(K key);

  // Returns the list of values V referenced by K key
  public abstract List<V> getAll(K key);

  // Stores a V value with the K key. If previous values exist, they
  // will be replaced
  public abstract void put(K key, V value);

  // Stores a list of V values with the K key. If previous values
  // exist, they will be replaced
  public abstract void putAll(K key, List<V> list);
}
```

The methods provided by class `DataManager` are only related to the storage and to the retrieval of data from the data manager. All of them require a key as parameter because the key is used to reference a value.

Method `put` stores a value with the given key into the data manager. On the other hand, method `get` retrieves the value from the data manager associated to the given key. Analogously, methods `putAll` and `getAll` allows to store/to retrieve a list of values associated to a given key respectively, instead of a single value. On the occasion of storing a value with a key that has already been used before, the previous stored values are replaced. And when calling method `get` but it is a list what is stored, the first element of the list is returned.

A session can dispose of more than one data manager. One could be used to store values of type `String` and another could be used to store values of type `Integer`, for instance. It is the

`dm_key` – the argument of the constructor of class `DataManager` – that distinguishes each of the data managers available in the session.

`DataManager` is flexible with the key type chosen by the user to store values. The key could be of any type (`Integer`, `String`, etc). An example of a session interacting with several data managers is illustrated in Figure 4.2.1.

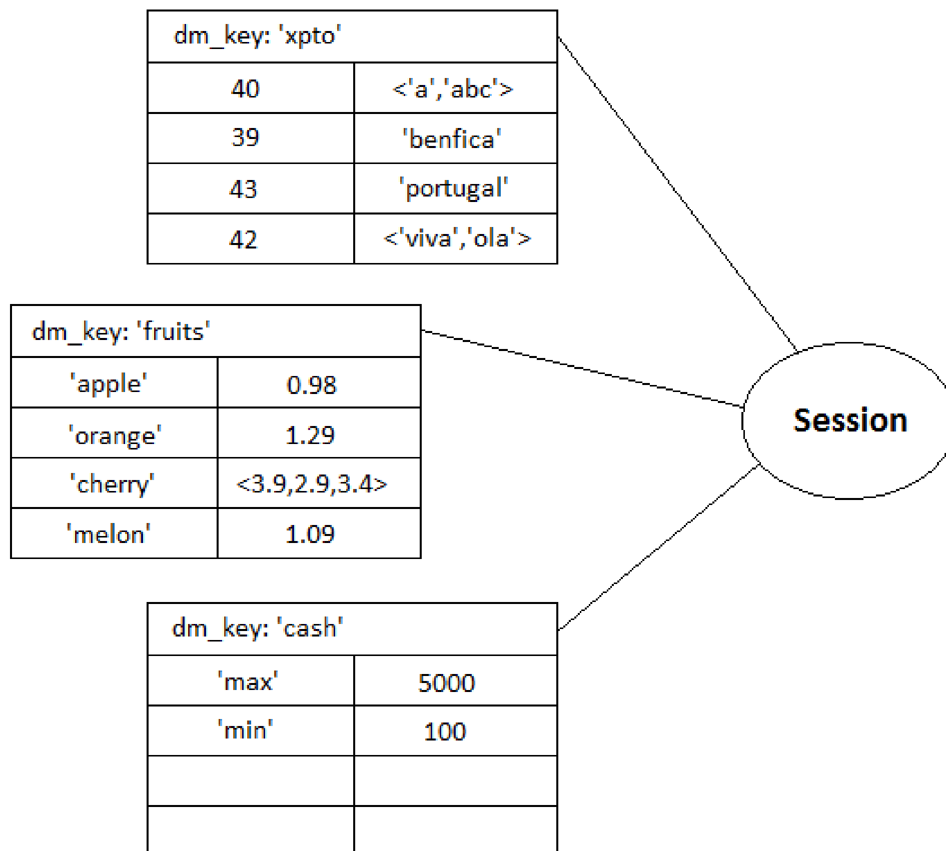


Figure 4.2.1 – A session interacting with several data managers

4.2.5. Itinerary

Class `Itinerary` offers an interface which allows the user to enhance the potential existing in the mobility of a session. One of the features is that it is possible to define the traveling strategy for a session – or a set of sessions – within the same itinerary. In Chapter 3, we presented two traveling strategies: unidirectional and multidirectional. In the instantiated model, the traveling strategies can be used individually or combined. Figures 3.3.1 and 3.4.1 from Chapter 3 represents examples of traveling strategies applied individually to an itinerary and Figure 4.2.2 is an example of both strategies combined.

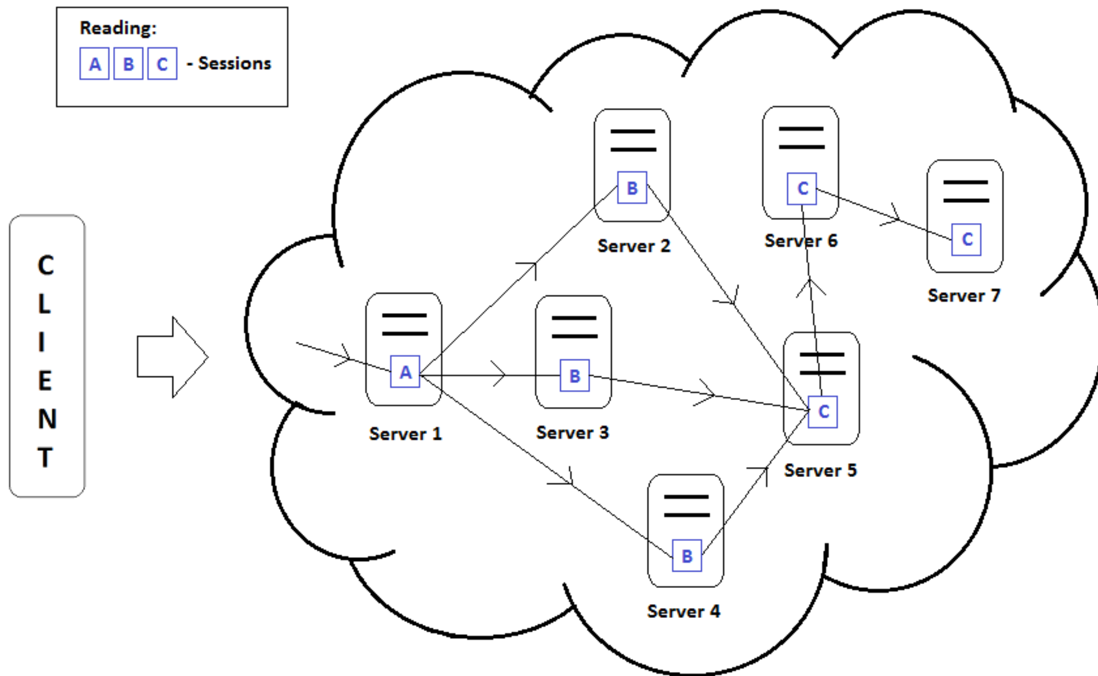


Figure 4.2.2 – Example of an upload with both traveling strategies combined

In Listing 4.2.6, it is described the public interface of class `Itinerary`:

Listing 4.2.6: The `Itinerary` class

```
public class Itinerary implements Serializable {

  // Adds a session to be uploaded to the location identified by server
  public <T> void add(Session<T> session, Server server) {…}

  // Adds a session to be uploaded to all the locations identified
  // by server. The traveling strategy for the session is
  // unidirectional.
  public <T> void addUniDirection(Session<T> session, Server server){…}

  // Adds a session to be uploaded to the locations identified
  // by the list of servers. The traveling strategy for the session is
  // unidirectional.
  public <T> void addUniDirection(Session<T> session, List<Server>
         serversList) {…}

  // Adds a session to be uploaded to all the locations identified
  // by server. The traveling strategy for the session is
  // multidirectional.
  public <T> void addMultiDirection(Session<T> session, Server
         server) {…}

  // Adds a session to be uploaded to the locations identified
  // by the list of servers. The traveling strategy for the session is
  // multidirectional.
  public <T> void addMultiDirection(Session<T> session, List<Server>
```

```
            serversList){…}

    // Uploads this itinerary to the network
    public Future<List<Result<?>>> upload() throws
            SessionNotFoundException, HomeCreationFailedException {…}

    // Sets a remote DataManager identified by url for this Itinerary
    public void setRemoteResultStorage(URL url) {…}

    // Adds a DataManager dm to this Itinerary
    public void addDataManager(DataManager<?,?> dm) throws
            KeyNotAcceptedException {…}

    // Returns a DataManager associated to the given key
    public DataManager<?,?> getDataManager(String key) throws
           KeyNotFoundException {…}

    // Bridges session1 to session2. Session2 must be of Unidirectional
    // traveling
    public <T> void bridge(Session<T> session1, Bridge<T> session2)
            throws BridgeException, SessionNotFoundException {…}

    // Bridges session1 to session2. Session2 must be of MultiDirectional
    // traveling
    public <T> void bridgeMulti(Session<T> session1, Bridge<List<T>>
            session2) throws BridgeException, SessionNotFoundException {…}
}
```

Method `add` simply adds to the current itinerary, a session to be deployed in a given server.

In regard of applying a traveling strategy for a session, there are two versions offered for each strategy. Similarly to method `add`, these methods add to the current itinerary, a session to be deployed in a given server or list of servers. Methods `addUniDirection` assign a unidirectional traveling strategy and methods `addMultiDirection` assign a multidirectional traveling strategy.

Once the itinerary has at least one session added – done by any of the methods introduced above – the user can call the method `upload` to dispatch this itinerary to the network. Similar to class `Session`, the method `upload` provided in class `Itinerary` returns a future list of results. Since the time expended by sessions to complete execution may vary greatly, the use of an asynchronous computation is more adequate. When calling the method `upload`, if the system was not capable to create the mechanism for the itinerary to deliver the computed results back home, a `HomeCreationFailedException` is thrown. In this situation, the session is not uploaded to the network.

In class `Itinerary`, the user is able to add a data manager to the itinerary through method `addDataManager`. If the key attributed to the data manager already exists, a `KeyNotAcceptedException` is thrown. Analogously, any of the stored data managers can be

retrieved by calling `getDataManager` with the associated key and in case the data manager doesn't exist, a `KeyNotFoundException` is thrown.

Figure 4.2.3 illustrates the setting by default: the data is stored locally and carried from machine to machine within the itinerary. This might be particularly useful when sessions compute large amounts of data and might not be intelligent to carry them from host to host within the itinerary. Figure 4.2.4 depicts the case where the method `setRemoteResultStorage` is used by the client to indicate that the itinerary must use a data manager located remotely to store and to retrieve data.

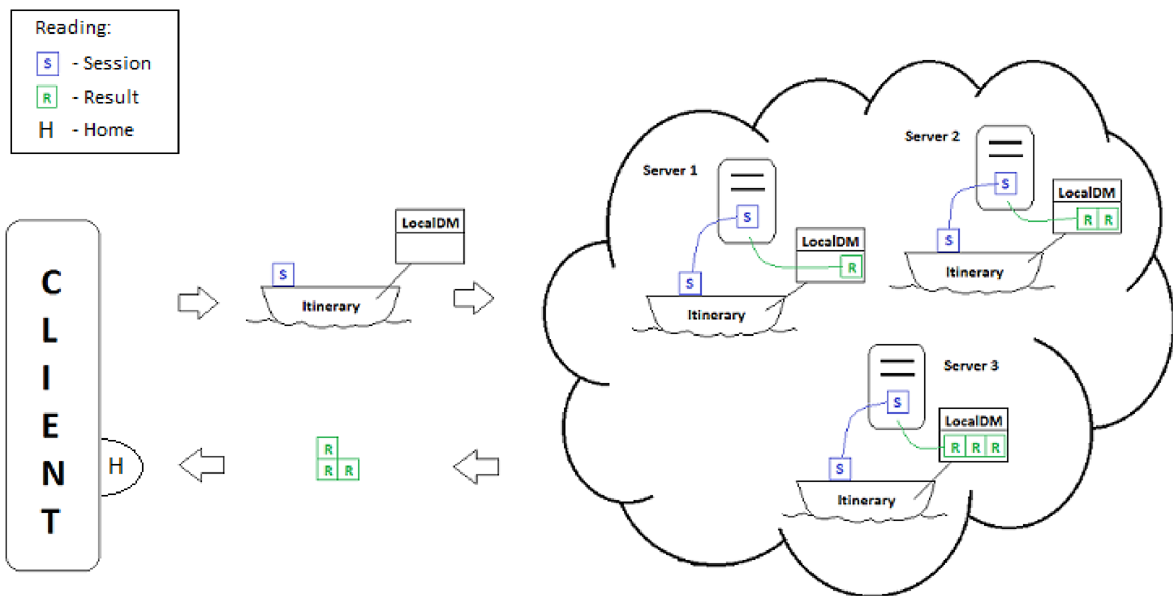


Figure 4.2.3 – Session using a local data manager to store results

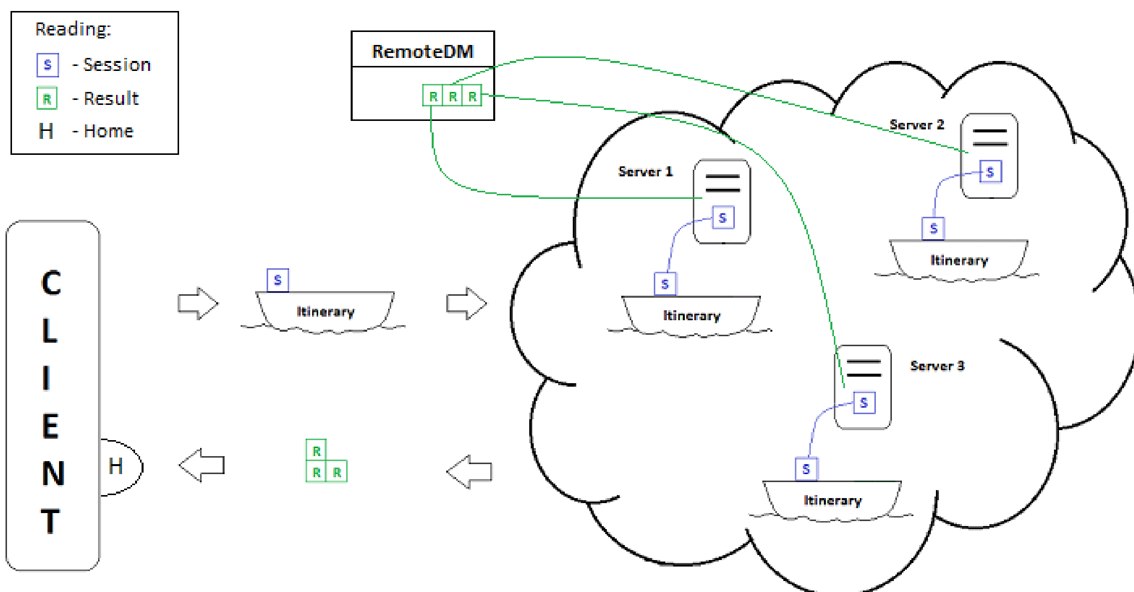


Figure 4.2.4 – Session using a remote data manager to store results

4.2.6. Bridging

Regarding the bridging of sessions, class `Itinerary` offers two methods: `bridge` and `bridgeMulti`.

To make a bridge between two sessions, the user-programmed session is required to implement interface `Bridge` (Listing 4.2.7). Then, similarly to interface `SessionRunnable`, the user is obliged to implement the only method specified in `Bridge`: `setArg`.

Listing 4.2.7: The `Bridge` interface

```
public interface Bridge<T> {
  void setArg(T arg);
}
```

On a bridging, the value given by the parameter `arg` in the method `setArg` will contain the results of the other session once it has finished executing. Thus, when writing the code for the session in method `run()`, the parameter `arg` of method `setArg` could be trivially used by the programmer.

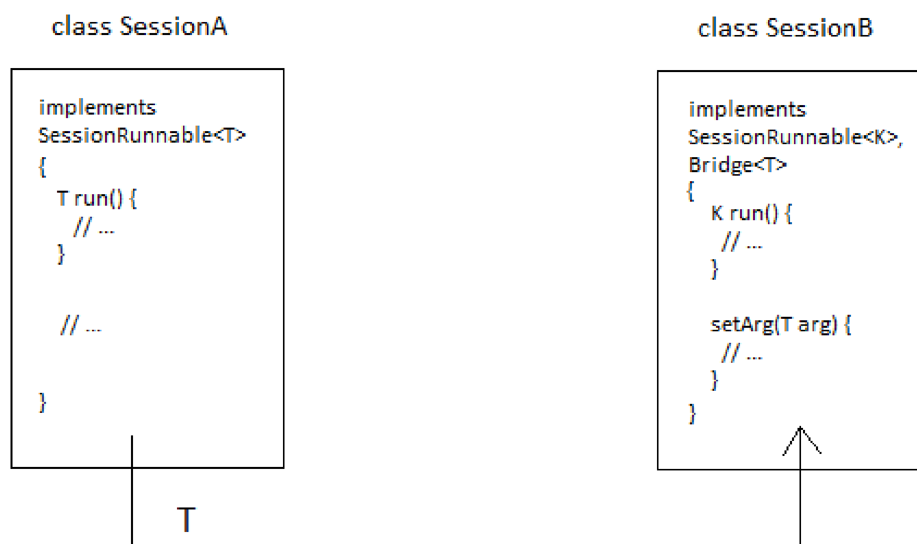


Figure 4.2.5 – Bridging between two sessions

A condition for a `bridge` to be successfully made (Fig. 4.2.5) is that the result computed by the first session must be of the same type of the parameter used in the `setArg` method. The design of our model allows the user to acknowledge of compile-time errors that may occur in case the code written by the user for the migrating-component is not correctly implemented. For instance, if a session returning a result of type `String` is bridged to a session that uses type `Float` as parameter in the `setArg` method, will cause a compile-time error. An additional condition to successfully bridge two sessions is that they must have been already added to the `Itinerary` prior to the bridge, otherwise a `SessionNotFoundException` is thrown.

An important requirement to successfully bridge two sessions is that the first session must use a unidirectional traveling strategy; otherwise a `BridgeException` is thrown. This is due to a session migrating to many servers – i.e. multidirectional – returns a list of results but the session that is bridging with it accepts only one. Thus, the bridging cannot be made. It is because of this situation that prompted the development of `bridgeMulti` which offers the same functionalities as `bridge` except that `bridgeMulti` requires the first session to use multidirectional traveling strategy rather than unidirectional. This way, the list of results could be copied to the other bridging session. These two methods are illustrated respectively in Figures 4.2.6 and 4.2.7.

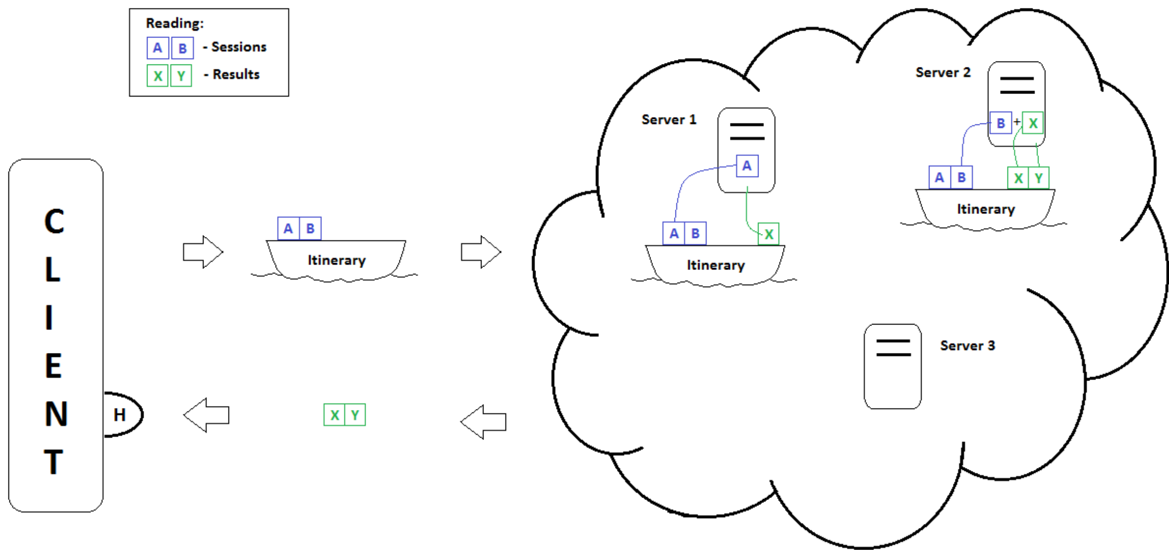


Figure 4.2.6 – Bridge in unidirectional traveling

Figure 4.2.6 represents two sessions A and B that has a bridge and uses unidirectional traveling- The result X computed by session A is used in the execution of session B.

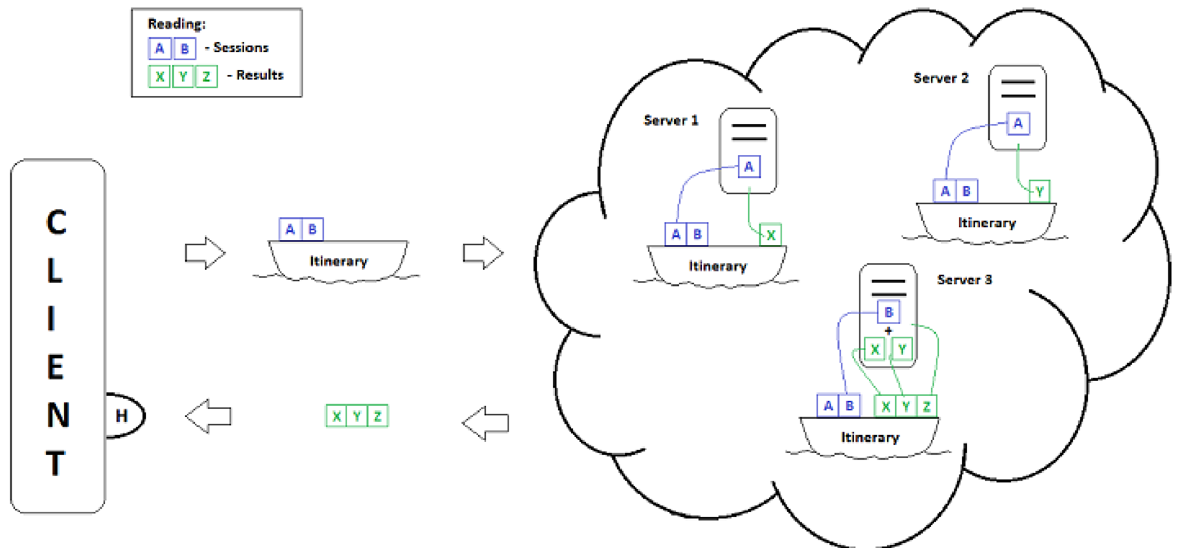


Figure 4.2.7 – Bridge in multidirectional traveling

Figure 4.2.7 illustrates two sessions A and B that has a bridge and uses multidirectional traveling. The results X and Y computed by session A are used in the execution of session B.

4.2.7. Code sample

The following four listings contain a code example for the scenario represented in Figure 4.2.2 where the traveling strategies are both combined in the uploading of `SessionA`, `SessionB` and `SessionC`. The bridging feature is also used between `SessionA` and `SessionB` as described in the code of class `SimpleItinerary`. Finally, Listing 4.2.12 contains the output of the computed results from this scenario.

It is important to notice that the migrating sessions are always JavaBeans. The reason behind this requirement is that Web services only accept Java primitive types (e.g. `int`, `byte`, `String`, `Object`, etc) to be passed as parameter in their Web methods.

Listing 4.2.8 – SessionA code

```
public class SessionA extends Session<String> implements Serializable{

  // it is the user that defines the type that is returned by the
  // method run
  public String run() {
     String str = "apple";  // just an example
     // …
     // any code written by the user
     // …

     return str;
  }
}
```

Listing 4.2.9 – SessionB code

```
public class SessionB extends Session<String>
  implements Serializable, Bridge<String> {

  private String arg;

  public String run() {
     String str = " is delicious";  // just an example
     // …
     // any code written by the user
     // …

     return arg + str;
  }

  // a class that implements Bridge must implement method setArg
  public void setArg(String arg) {
    this.arg = arg;
  }

}
```

Listing 4.2.10 – SessionC code

```
public class SessionC extends Session<Integer> implements Serializable{

  public Integer run() {
     Integer myInt = new Integer(10); // just an example
     // …
     // any code written by the user
     // …

     return myInt;
  }
}
```

Listing 4.2.11 – Example of a class using `SessionA`, `SessionB` and `SessionC` with both traveling strategies combined

```
public class SimpleItinerary {

  public static void main(String[] args) {

      SessionA sessionA = new SessionA();
      SessionB sessionB = new SessionB();
      SessionC sessionC = new SessionC();

      Itinerary it = new Itinerary();

      // Server1 to N could be for instance a ServerURL("http://…")

      // sessionA is defined to go to Server1
      it.addUniDirection(sessionA,Server1);  // unidirectional travel
```

```
        // bList contains the list of servers for SessionB to migrate to
        List<Server> bList = new ArrayList<Server>();
        bList.add(Server2);
        bList.add(Server3);
        bList.add(Server4);
        it.addMultiDirection(sessionB,bList); // multidirectional travel

        // cList contains the list of servers for SessionC to migrate to
        List<Server> cList = new ArrayList<Server>();
        cList.add(Server5);
        cList.add(Server6);
        cList.add(Server7);
        it.addUniDirection(sessionC,cList);  // unidirectional travel

        // bridging
        try {
              it.bridge(sessionA,sessionB);
        } catch (BridgeException e) {
              e.printStackTrace();
        }

        // upload itinerary and wait asynchronously for the results
        Future<List<Result>> future;

        try {
              future = it.upload();
        } catch (SessionNotFoundException e) {
              e.printStackTrace();
        } catch (HomeCreationFailedException e) {
              e.printStackTrace();
        }

        List<Result> results;

        try {
              results = future.get();
        } catch (InterruptedException e) {
              e.printStackTrace();
        } catch (ExecutionException e) {
              e.printStackTrace();
        }

        // printing the results
        for (Result<?> r : results) {
          System.out.println("Owner: " + r.getOwner());
          System.out.println("Result: " + r.getResult() + "\n");
        }
    }
}
```

Listing 4.2.12 – Returned results

```
Output>

        "Owner: Server1";
        "Result: apple";

        "Owner: Server2";
```

```
"Result: apple is delicious";

"Owner: Server3";
"Result: apple is delicious";

"Owner: Server4";
"Result: apple is delicious";

"Owner: Server5";
"Result: 10";

"Owner: Server6";
"Result: 10";

"Owner: Server7";
"Result: 10";
```

In this section, we introduced the API. In the following sections, we will present the middleware architecture and its implementation in order to comprehend the processes running 'inside' the middleware that are required to accomplish the requests submitted by the client.

4.3. Middleware Architecture

This section describes the middleware which is responsible for the management, transportation and execution of user-programmed components dispatched to the network.

As mentioned in the beginning of this chapter, all communication between remote processes resorts to Web service technology, i.e., to Web standards enabling migration of sessions across the Internet.

The process of sending a session to a remote machine requires several stages. Thanks to the interface, the user is not aware of the operations done in these stages. To the user, the only requirement is to call the method upload to perform the sending process. In regard of the middleware, it has to ensure that the user-programmed session is successfully executed remotely and the computed results are successfully delivered to the respective client. In other words, the middleware is responsible for the migration of sessions in the network and this requires embodying the traveling component of requisites to successfully execute in remote machines. These operations are fundamental not only to hide from the user the complexities in the process of migrating sessions – and obtaining their results – but to give total control to the middleware whose job is to satisfy the demands made by the client application.

To dispose of a middleware layer that allows migration, execution and returning of sessions results, it was necessary to embody the middleware of functionalities that cooperate with each other at the client side and at the server side.

Figure 4.3.1 shows the middleware architecture sliced between the client and the server, displaying the components existing at each side. In order to make possible the communication between remote hosts, Web services – represented in the figure with the acronym WS – were assigned to make the bridge between components residing in different machines.

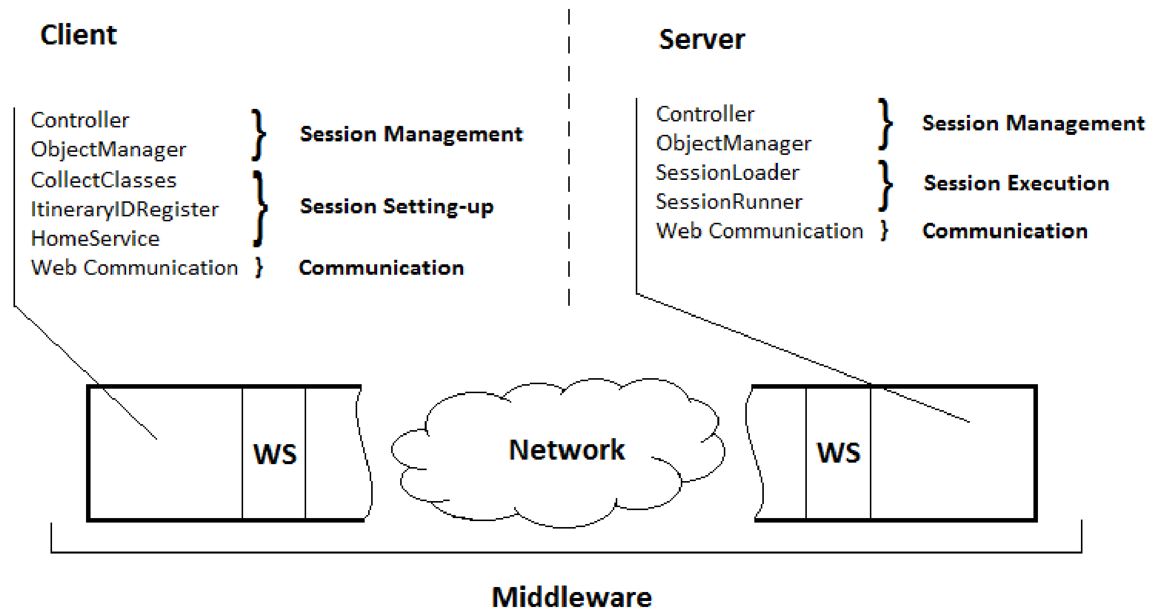


Figure 4.3.1 – Middleware architecture

Next, we are going to explain each of these middleware functionalities in four different categories: Session Setting-up, Session Execution, Session Managing and Communication.

4.3.1. Session Setting-up

Functionalities belonging to this group have the responsibility of setting up user-developed components in order to successfully execute in any remote machine. The setting-up process is done prior to the sending of the session to the first host. The idea is to include within the session, information that is necessary for its trip (Fig. 4.5.3) not only to successfully execute in remote machines but also to be able to deliver the computed results back to the client. Session setting-up components reside only in the client machine.

CollectClasses is a component responsible for finding and collecting all classes that are required to execute the user-programmed session.

HomeService is responsible for receiving the computed results from sessions that have completed their trip and to deliver the results to the client application.

ItineraryIDRegister attributes a unique ID for each session or itinerary dispatched by the user. In other words, all user-developed components have an ID assigned by the middleware in order to have control of all sessions and itineraries that are running in the network.

4.3.2. Session Execution

Components belonging to this category are in charge of putting sessions into execution once they arrive to the server. Session execution components are located only in the server machine.

SessionLoader is responsible for loading the classes required to run the session on the local JVM. These classes were collected on the client side by *CollectClasses* component presented above.

SessionRunner is the component assigned to run sessions. When a session arrives to a given server, *SessionLoader* loads the classes to the JVM and then, *SessionRunner* executes the session.

4.3.3. Session Management

Components belonging to this group have the responsibility of managing sessions sent to the network. Most of these components are needed in the client and server machines because they are required by other components to do their job.

ObjectManager is responsible in transforming an object into an array of bytes and in transforming an array of bytes into an object. The former functionality is used for example to transform a migrating session into an array of bytes before it is sent to a given host through a Web service method. The inverse operation is done for example when the session arrives to the server.

Controller is a component consisting of operations related to the upload of user-developed components across the network, and to the returning of computed results to the client. Although the sending process is done through the Web methods provided by the stubs, it is the *Controller* that checks where the session is set to move next. In case a session has completed its trip, the *Controller* grabs the computed results and sends them to the *HomeService* component. Moreover, *Controller* is also in charge of checking the traveling strategy assigned for a session in order to call an operation to send the session in parallel or an operation to send it in sequence.

4.3.4. Communication

Communication components can be resumed to the stubs that are created from the WSDL document of a Web service. In our model, we used the *wsimport*[1] tool to generate client-side run time classes which provide mechanisms for message exchanging between components running in different hosts. In other words, the generated stubs offer Web methods which are used by the middleware components to pass data to remote stations. For instance, component

[1] tool that generates JAX-WS portable artifacts, such as Service Endpoint Interface, Service, Exception class mapped from wsdl:fault, Asynchronous Response Bean derived from response wsdl:message and JAXB generated values types (https://jax-ws.dev.java.net/jax-ws-ea3/docs/wsimport.html)

Controller uses a Web method provided by a specific stub to transfer a session from the current machine to another.

4.4. Itinerary Lifecycle

The objective of this section is to offer a visual perspective of our model in order to understand the chain of events happening 'inside' the middleware when the user uploads a session to the network.

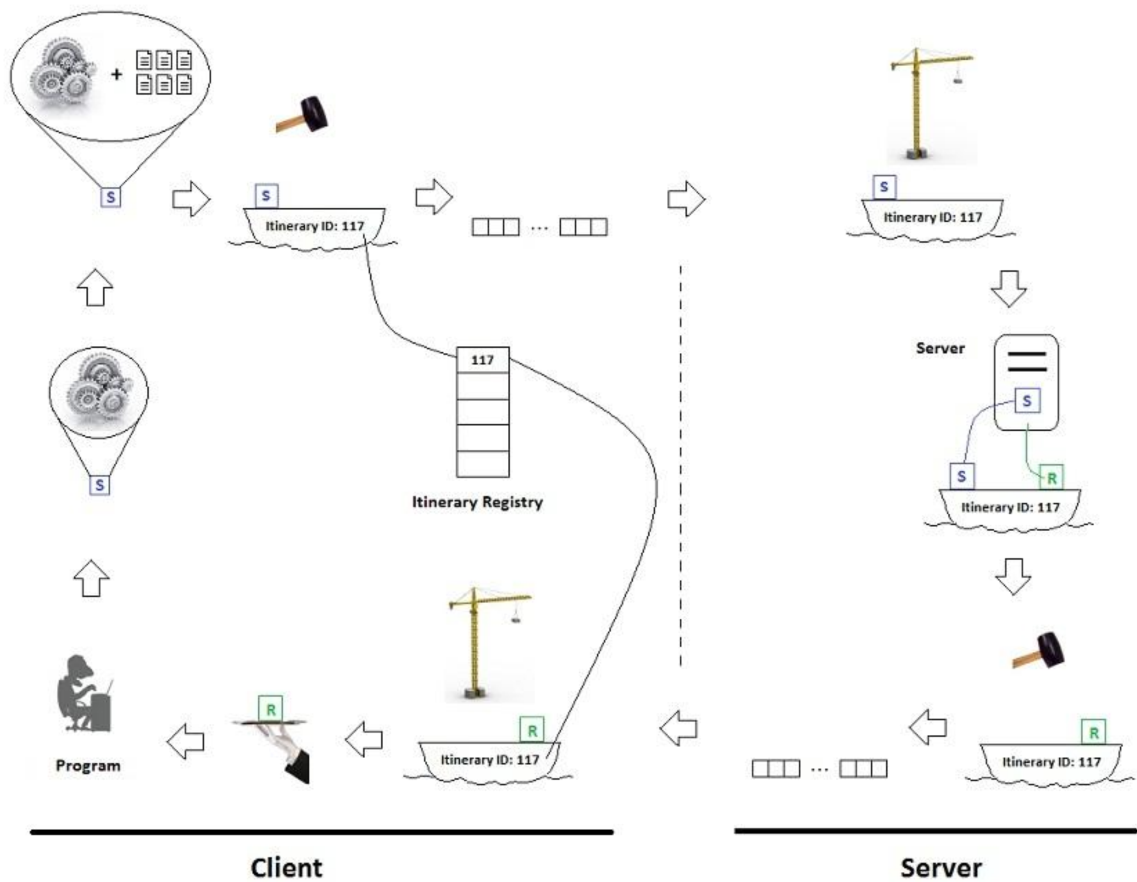


Figure 4.4.1 – Lifecycle of an itinerary

Figure 4.4.1 shows a schema representing the lifecycle of an itinerary. Starting in the program, the sequence demonstrates the applicability of each component working in the client and in the server machines until the computed result is delivered to the program. Each of these components have a well defined task in order to make possible the uploading of a session, the execution in any remote computer and the return of computed results.

Let's explain the sequence in the figure: starting in the program (lower left corner) that uploads a session, the *CollectClasses* component looks for the user-programmed classes needed to execute the session and packs them together with the session. On the next stage, the

middleware attributes an itinerary with an id (117 in the figure) to hold the session and registers it with the *ItineraryIDRegister.* Next, the *Controller* 'informs' the itinerary about the location of the client machine in the network and checks if the *HomeService* is active. If not, *HomeService* is launched in the client computer. Then, the *ObjectManager* smashes the itinerary into an array of bytes and the *Controller* uploads this array of bytes through a Web method provided in the stubs components.

On the server side, the *ObjectManager* rebuilds the itinerary component that has arrived in the form of an array of bytes. When the itinerary is reconstituted, *SessionLoader* loads the classes required to execute the session to the local JVM. Once they are loaded, *SessionRunner* component runs the session that is defined to be executed in this server. When the execution has completed, the *Controller* verifies that the itinerary – now holding a result – "wants to go home". Thus, *ObjectManager* is called to transform the migrating component into an array of bytes to be sent by a Web method given in the stubs.

In the last procedure, now back to the client machine, the *ObjectManager* rebuilds the itinerary and the *Controller* looks in the itinerary registry table to acknowledge which itinerary has arrived in order to correctly deliver the computed results to the programmer.

Now that we have acknowledged of the activities happening 'inside' the middleware, we proceed to the explanation of the middleware implementation in the next section.

4.5. Middleware Implementation

Throughout this work, we have generally assumed that the user–programmed components are of type `Session`. It is important to recall that components of type `Itinerary` could also be uploaded by the user as denoted in the beginning of Section 4.1. A user-programmed component of type `Itinerary` contains one or more components of type `Session` with routes defined by the user and features that enhances its mobility in the network.

4.5.1. User-developed components transformation into middleware components

It is important to understand that the itineraries presented in the figures with the shape similar to a 'vessel' are not the same itineraries programmed by the user. This 'vessel' which is able to travel in our model is constructed by the middleware to transport user-developed sessions to their destinations. The repository of classes needed to execute sessions remotely are also included by the middleware in the 'cargo' of this itinerary. In order to distinguish the 'vessels' traveling in the instantiated model, each of them has a unique identifier assigned by the middleware before the departure from the client machine.

In fact, the class that constructs the middleware `Itinerary` is not much different from the API class `Itinerary`. The former is equipped with more mechanisms which give total control to

the middleware at each stage of its lifecycle. In case the user uploads a component of type `Itinerary`, the 'cargo' – which consists of sessions – of this component is simply transferred to the internal `Itinerary` constructed in the middleware. The reason behind the implementation of an internal `Itinerary` is to have those mechanisms – or methods – available anywhere in the middleware but hidden from the user.

Similarly to an `Itinerary` uploaded by the user, a component of type `Session` is also 'transformed' into a middleware `Session`. The reason behind this procedure is the same: the system cannot afford a client to call methods that can interfere with a component that is already in the middleware.

So, components of type `Session` and of type `Itinerary` uploaded by the client are 'transformed' into a middleware `Session` and `Itinerary` respectively. In Figure 4.5.1, we have an image showing the data held in each of them.

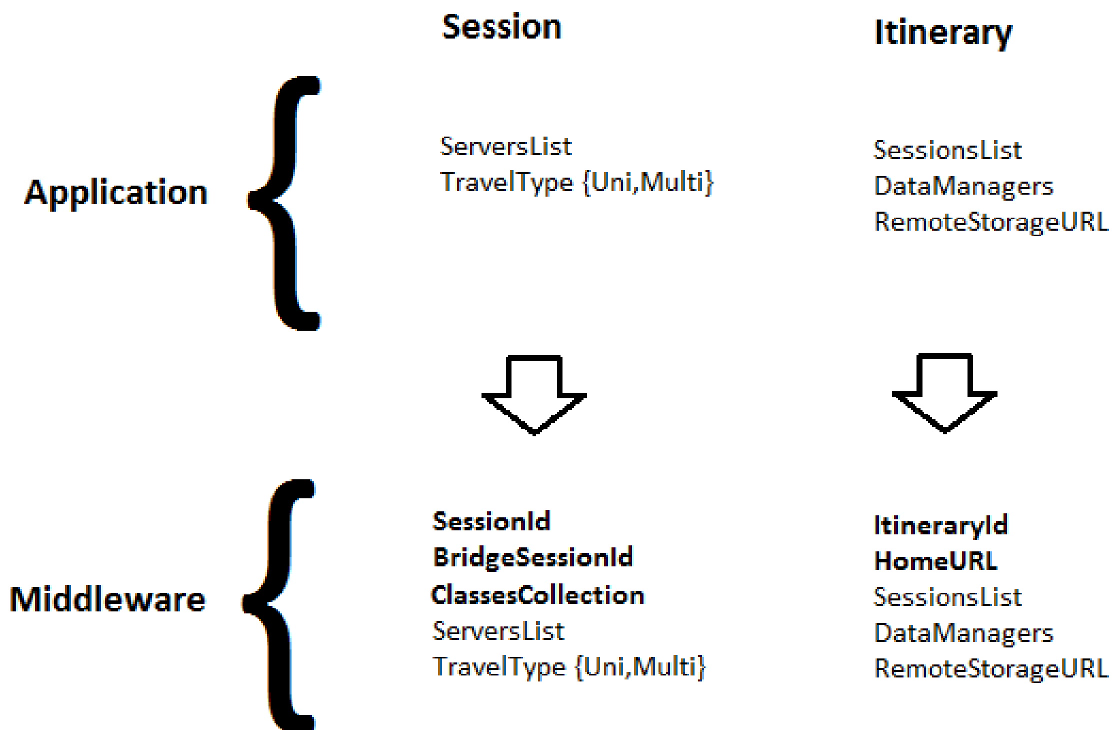


Figure 4.5.1 – `Session` and `Itinerary` contents in the client and in the middleware

In the client application, `Session` and `Itinerary` contents are defined by the user. In the middleware, `Session` and `Itinerary` contents comprises of data added by the middleware (in bold) and the information indicated by the user.

Client Session Contents

ServersList is the list containing 1 to N servers for this session to migrate to.

TravelType is a flag that represents the traveling strategy applied for this session. If the user created a session without specifying the traveling strategy, the middleware applies by default a unidirectional strategy for the session. It is important to understand that this flag cannot be affected by the user if it is a single session that is uploaded (API class `Session` does not offer any method to define the traveling strategy for a session). In other words, the traveling strategy can only be specified to a session through class `Itinerary`. The itinerary which holds the session is then, responsible to transport it to the destination(s) according to the traveling strategy specified by the user.

The reason behind not offering methods in API class `Session` for the user to specify the traveling strategy is because the migrating component is a single session only. However, since every user-developed component is transferred into a middleware `Itinerary` before being dispatched to the network, the traveling strategy has to be mandatorily specified in the background by the middleware. Thus, in class `Session`, the middleware applies a multidirectional traveling strategy for the session in methods `uploadAll` whilst in `upload`, the middleware 'obligates' the session to use a unidirectional traveling strategy.

Client Itinerary Data

SessionsList is a list containing 1 to N sessions that will be transported by this itinerary to their respective destinations.

DataManagers contains 0 to N data managers defined by the user.

RemoteStorageURL is a variable that specifies a remote data manager for the itinerary to store the computed results while it is migrating from host to host.

Middleware Session Data

SessionId is a unique identifier generated by the middleware for this session.

BridgeSessionId is the identifier of the session which is making a bridge with this session. In case there is no bridging, this value is null.

ClassesCollection is a repository of classes which the session requires to execute remotely. It is the middleware that looks up for these classes in the directory of the class that called the upload method. Classes that are in its subdirectories are also collected to be included within the middleware session.

SessionId and *BridgeSessionId* are required only to perform a bridge. In case the user calls a method to bridge a session to another, these identifiers are used by the middleware to copy the computed results from one session to the other.

Middleware Itinerary Data

ItineraryId is the identifier that represents this itinerary in the middleware. This id is used to distinguish the 'vessels' that are 'navigating' in our model. In Figure 4.4.1, the *ItineraryId* is 117.

HomeURL contains the address of the client machine which is required to deliver the computed results once the itinerary completes its trip. This information is essential otherwise the itinerary won't know where to return the computed results.

It is important to realize that the middleware session is the component that is executed in remote hosts and the middleware itinerary is the component that transports the sessions in the network.

In order to have a clearer perception of this 'relationship', Figure 4.5.2 allows us to understand that the middleware itinerary contains a list of middleware sessions. And the session that was in fact developed by the user is stored 'inside' this middleware session, along with the classes' collection and with the servers specified to execute it.

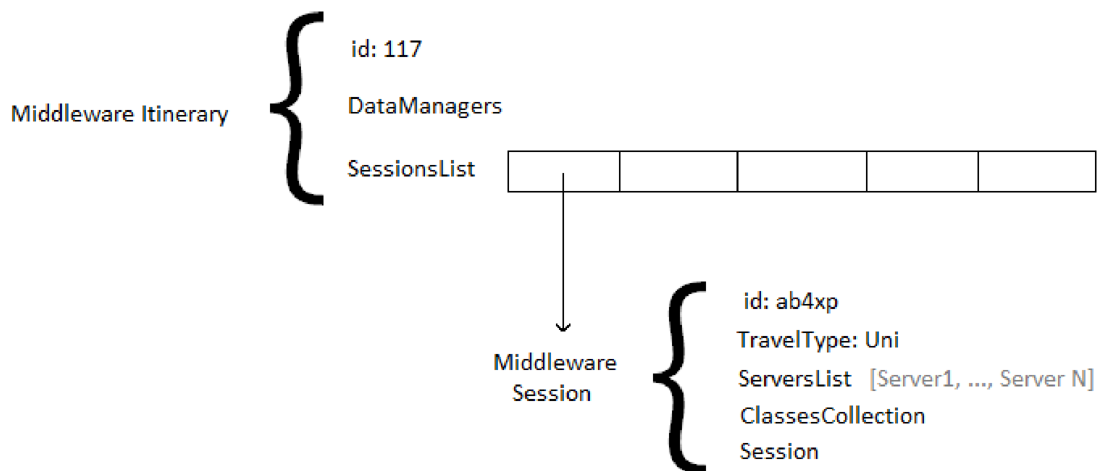


Figure 4.5.2 – Middleware itinerary

4.5.2. Itinerary departure procedures

Prior to the departure of the itinerary, the middleware generates a unique identifier for the traveling component and stores it in the itinerary registry map. This procedure is required because a user may upload two or more itineraries and thus, the id stored in the table is used to know which returned results corresponds to which itinerary that has been sent. In addition, a handler of the client thread responsible for the uploading is also stored in the same table and is associated with an itinerary id. The objective of the handler is to provide a way to synchronize the thread that receives the computed results with the thread of the client application that is waiting for them. Thanks to class `Future` (`java.util.concurrent`), the coordination

between these two processes is done asynchronously. This means that the handler stored in the itinerary map acts only when the results from the same itinerary id have returned.

Moreover, the middleware has to ensure that the client machine has a way to receive computed results sent from any remote machine. The component *HomeService* presented in Section 4.3 is the one that is responsible to receive the results by disposing a Web service (Listing 4.5.1) in the client machine. Thus, the middleware has to ensure that this Web service is enabled when an upload is made.

Listing 4.5.1 – client side Web interface

```
public class ClientWebService {
      void receive(SessionResults results) {…}
}
```

A task that is also in charge of the middleware is to create a data manager to store computed results. By default, a local data manager is generated as part of the 'cargo' of an itinerary. Thus, it will travel along with the itinerary. Nonetheless, our model allows the user to specify a remote storage recipient to be used. When this option is chosen, the local data manager is not generated and the itinerary is defined to use the remote storage facility to save computed results. This particular data manager is defined with the key "RESULTS_DM" and a `KeyNotAcceptedException` is thrown if the user attempts to add a data manager with this key.

Once the vessel is set to depart, the first destination to migrate to, is the first server specified in the list of the first session in the list. But before uploading the session, the middleware has to verify the traveling strategy applied by the user. If the session is set to travel with unidirectional strategy, then it is uploaded 'normally' to the first server. In case the session is set to travel with multidirectional strategy, then the middleware has to dispatch a copy of the itinerary to each of the servers defined to run the session.

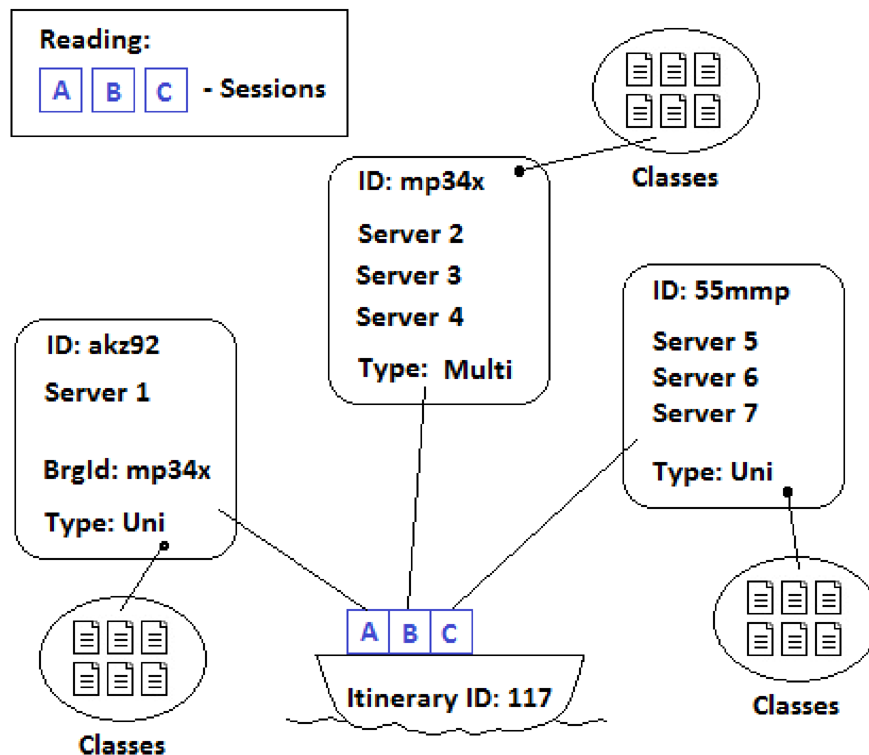


Figure 4.5.3 – Example of an itinerary before leaving the client machine

The itinerary represented in Figure 4.5.3 can be interpreted as the same itinerary of Figure 4.2.2 with its code written in Listing 4.2.11 in the moment before leaving the client computer. In this image, we can see that the 'vessel' is identified with id 117 and has three sessions: session A is defined to travel "unidirectionally" to sever 1; session B is specified to go to servers 2, 3 and 4 "multidirectionally", and finally, session C is defined to migrate "unidirectionally" to servers 5, 6 and 7. The classes' collection is also part of the 'cargo' of the itinerary component.

Each session also has a unique identifier within the itinerary attributed by the middleware. This identifier is needed for the bridging mechanism. In other words, the identifier is used by the middleware to pass the results computed by a session to one that has a bridge with it. It is the identifier that distinguishes the sessions that are in the list within the itinerary. Some sessions may have a bridge, some may not. In the example displayed above, session A is making a bridge with session B (proved by the *BrgId* variable). Thus, when session A completes its execution in server 1, the computed result is copied to session B.

The act of uploading is done through a Web method called *transfer* (Listing 4.5.2) and is defined in the stubs components. Since Web services only accept Java primitive types (e.g. `int`, `byte`, `String`, `Object`, etc), the migrating itinerary – and its constituents – is required to be transformed into one of these types.

Listing 4.5.2 – server side Web interface

```
public class ServerWebService {
      void transfer(SessionTransfer session) {…}
}
```

Listing 4.5.3 – `SessionTransfer`

```
public class SessionTransfer {
  byte[] code;
  String mainClass;
  byte[] object;
}
```

A class called `SessionTransfer` (Listing 4.5.3) was created with the objective of representing the migrating component which contains the itinerary and its constituents. A `SessionTransfer` consists of three important variables: the name of the session class (type `String`) that is going to run remotely, the code of the classes required to run the session (type `byte`) and the itinerary itself (type `byte`). This way, our model offers a type of component that is accepted to be used in the Web methods.

Thanks to *ObjectManager*, the itinerary is transformed into an array of bytes and the classes of the session that is going to execute, are also transformed into an array of bytes. These two variables are set into a `SessionTransfer` component together with the name of the main class. Then, the upload is effectively done as the `SessionTransfer` component is transferred from the current machine to a remote host.

4.5.3. Itinerary server arrival procedures

Once the `SessionTransfer` component arrives to a server, the array of bytes containing the code of the classes of the session that is going to execute, is rebuilt into an object thanks to the *ObjectManager* component. Then, the *SessionLoader* adds these classes to the local JVM.

In regard of the array of bytes containing the itinerary, the process is not exactly the same as done with the classes' collection. In this case, the *ObjectManager* component has to work together with the *SessionLoader*. The reason for this is because the itinerary object can only be reconstructed while the corresponding class is loaded from the JVM.

Finally, the last variable stored in `SessionTransfer` component is the name of the class that is going to be executed. Otherwise, the server wouldn't be able to select from the list, which is the session class to run.

From this moment, *SessionRunner* component entries into action: it runs the session and stores the results in the "RESULTS_DM" data manager. Next, it checks if the running session has a bridge with another one. In case there is a bridging id found in this session, then the 'freshly' computed results are copied to the other session to be used in its calculations.

Still in the server machine, the next stages are performed by the *Controller* component. It checks if the current session has more servers to migrate to: if yes, then the vessel is transferred to the next server defined in the list. Otherwise, the *Controller* checks if there are more sessions in the itinerary.

If there are no more sessions waiting to be executed, then the itinerary has completed its trip and delivers the results to the client. Otherwise, the *Controller* acknowledges the location of the first server of the following session and transfers the itinerary to this machine.

Additionally to the processes explained above, the *Controller* is also programmed to check the traveling strategy specified in the session before transferring the itinerary to a remote station. When the strategy is unidirectional, the itinerary can be 'integrally' dispatched to the next server. Otherwise – a multidirectional traveling is thus defined – a copy of the itinerary is sent to each server stated in the session. In this last situation, the thread responsible for transferring the 'vessel' to each of the servers has to wait for the computed results to return. This process is required to synchronize the computed results which are delivered from different machines. Thus, the 'vessel' with id 117 defined in Figure 4.5.3 will take a route similar to the one represented in Figure 4.5.4.

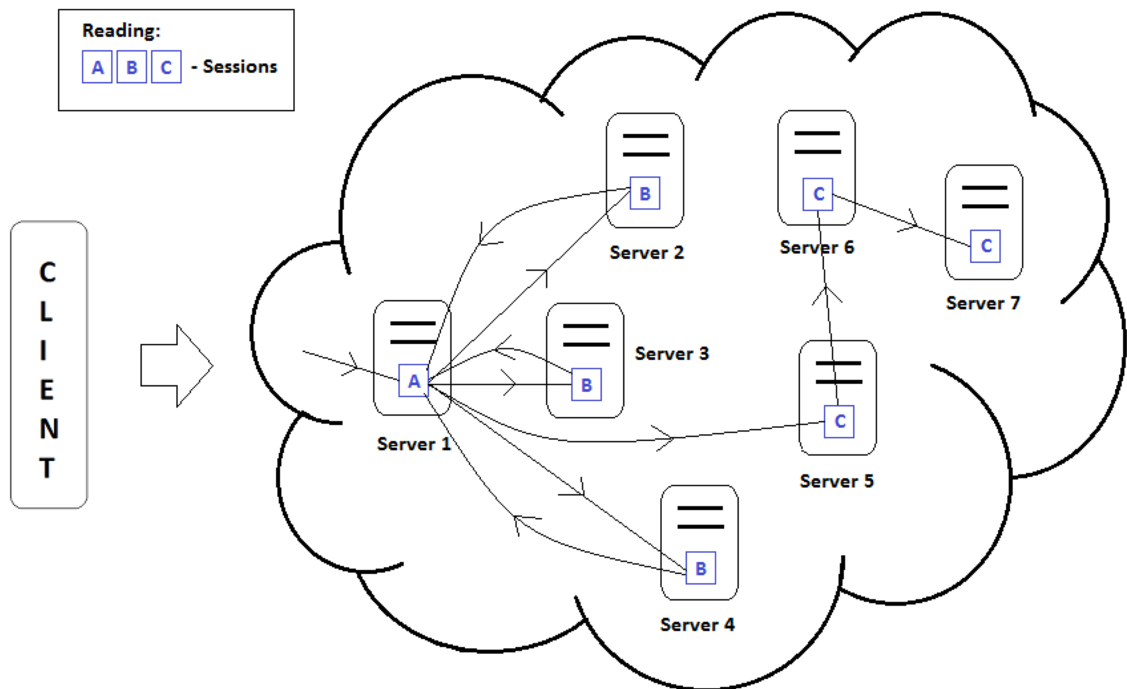


Figure 4.5.4 – Real traveling route of itinerary 117

Notice in Figure 4.5.4 that after session A has finished its execution, a copy of the itinerary is sent to Servers 2, 3 and 4 with the objective of running the next session, which is session B. But the thread responsible for the sending in Server 1 waits for the three results to return. Only after the last result has returned that the itinerary migrates to Server 5 in order to execute session C.

No matter if the user-developed component travels unidirectional or multidirectional, the middleware uses asynchronous communication in both situations. From the first upload triggered by the client until the upload to the last server, our model uses only asynchronous computation/communication. Thanks to class `Future` (`java.util.concurrent`) which

represents the result of an asynchronous computation, the middleware uses this class to receive the returning results that may arrive at any time. Looking carefully to the code sample provided in Listing 4.2.11, the upload method called by the client also returns a `Future`. Thus, the thread in the client computer also waits passively or asynchronously for the results to return.

In fact, the itinerary only leaves a host when the working session returns a result. In case the session has no more locations to go, it is simply offloaded from the itinerary after the result is stored in the dedicated data manager (RESULTS_DM). This means that the session and all its contents (e.g. the classes that belonged to this session) are discarded from the itinerary. Otherwise, the session is kept in the itinerary until the last server of its list is visited.

Hereafter, the logic is the same: the destination that the itinerary migrates to is always defined by the first session in the list of sessions. The following session – if any – takes place only when the first one has finished executing in all servers specified in its list of servers. If the list of sessions is empty, it means that the itinerary has finished its voyage and the results are sent to the client machine.

The process of migrating to the next machine is the same as explained in the beginning of this section: the *ObjectManager* transforms the itinerary into an array of bytes for a Web method (Listing 4.5.2) to transfer it to the next machine. After arriving to the next host, the process of rebuilding the itinerary, loading the session classes to the JVM and running the session, repeats again but this time in a different machine.

However, there is a situation that this process is different: it is when the last session finishes its execution in the last server. To be more precise, it is when the result computed by the last session in the last server is stored in the results data manager. This means that the vessel never travels back to the client machine. Since the client is only interested in obtaining the results, the contents returned are resumed to a list of results with the itinerary id that was responsible for them. The vessel that has been representing the itinerary in the figures, 'disappears' at this point.

The process of sending the object containing the list of results with the identifier of the itinerary to the client machine is similar to the one used in the itinerary sending to a server: the *ObjectManager* component transforms the object into an array of bytes and calls a Web method provided to transfer it (Listing 4.5.1). Nevertheless, like in all uploading processes in the middleware, the *Controller* component performs one crucial task before the itinerary is discarded: the "HomeURL" reference stored in the itinerary is labeled into the component that is migrating to the client machine.

4.5.4. Results returning to client

It is responsibility of the *HomeService* functionality to receive the returned results. After the component containing the list of results and the id of the itinerary has arrived to the client computer, the middleware uses the id to compare with those stored in the itinerary registry map

to obtain the associated handler. Once found, the handler is 'awaken' to handle the results from the *HomeService* component to the thread in the client application that is waiting to receive them.

The handler is a `Callable` (`java.util.concurrent`) that is created and executed in the middleware when the user uploads an itinerary and it is stored in the itinerary registry map. The task of the handler is to call the upload method to send the itinerary to the network and to wait for the results to return. In other words, the current thread responsible for the uploading call waits until another thread notifies the handler (`java.lang.Object`). When *HomeService* component receives the computed results, these are sent to the respective handler. It is this thread which is responsible for the sending that notifies the handler to return the results to the client application. Like the thread that is waiting in the client application for the results (the method `upload` returns an instance of class `Future`), the wait and notify mechanism of `java.lang.Object` is also asynchronous.

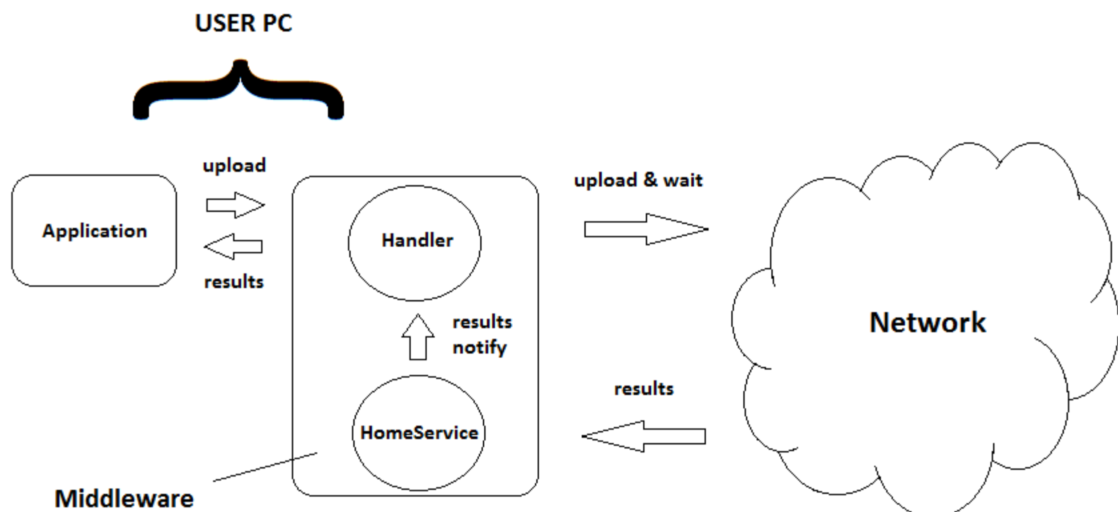


Figure 4.5.5 – Handler synchronization

In this chapter, the instantiation of our model has been described. In Chapter 5, we will present tests and the respective results performed in our model.

5. Evaluation

This chapter presents the evaluation of tests performed to our model and has the objective to understand the impact of session migration in comparison to remote interaction. The environment where the tests were conducted is a computer that worked as client and as server.

Figure 5.1 illustrates a session migrating to a server in order to interact locally (in the server) whilst Figure 5.2 shows a session executing in the client machine and interacting with a remote server.

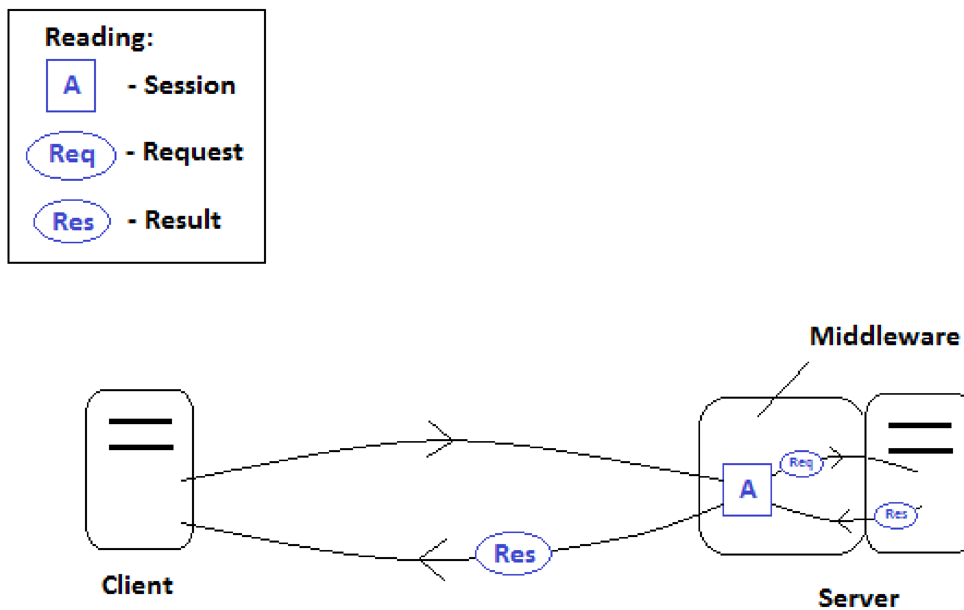
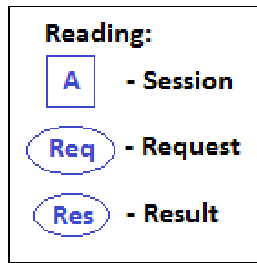


Figure 5.1 – Session executing in the server and interacting in the server



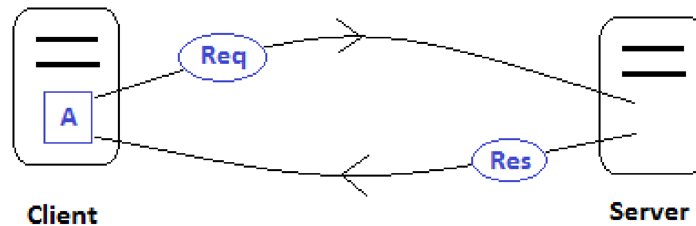


Figure 5.2 – Session executing in the client and interacting with a remote server

5.1. Environment Specifications

The presented values were measured on a laptop computer with the following characteristics: CPU Intel Core 2 Duo Processor T5500, 1GB memory DDR2 SDRAM 667Mhz, hard disk with 120 GB capacity and the operating system is Windows 7 Home Edition. All the values concerning network transfer times were estimated.

All tests were executed from within the eclipse framework and the Web server that hosts the Web services is Apache Axis.

5.2. Measuring the overhead of the middleware

This section presents the measurement that the middleware takes on average[2] in the uploading of a session to the network (Table 5.2.1). In order to not have inconsistent measurements, we closed all user-processes that were running in parallel in the system before proceeding with the evaluations.

	Message creation (sec)	Collect Classes (sec)	Smash Object (sec)	Build Object (sec)
1 class	0,047	0,005	0,017	0,032
10 classes	0,056	0,010	0,019	0,033
11 classes	0,059	0,011	0,019	0,033

Table 5.2.1 – Average time values in session uploading

[2] the average for each value was calculated by obtaining the mean of 20 tests

In the standards defined in Table 5.2.1, we considered three cases: one that a session consists of 1 class (2,059 KB), another consisting of 10 classes (20,59 KB) and another which contains 11 classes (22,645 KB). The message creation column refers to the time that the middleware requires to generate the message to be sent. Collect Classes' one indicates the time that the system requires to find and collect the classes of the session. Smash object column describes the time needed to transform an instance of class `SessionTransfer` (Listing 4.5.3) component into an array of bytes and build object's one refers to the time required to do the reverse operation, i.e., to rebuild the instance of class `SessionTransfer` from the array of bytes.

Moreover, another parameter that we had to consider in our calculations is the overhead of the SOAP envelope in the messages exchanged with a Web service. The tool we used to obtain the overhead value is called *tcpmon.* This program simply acts as a monitor between the client and the server to detect the content-length of transmitted messages (and also other characteristics but they are not required in our study).

5.3. Remote interaction versus remote execution (lower bandwidth)

The first evaluation conducted compares the temporal differences between a session executing in the client machine and performing one remote interaction with a given server and the migration of that same session to the server to perform the interaction locally. In this initial setting we assume a network with a bandwidth of 1 Mbps (125 KB/sec).

5.3.1. Remote interaction

We begin by presenting in Table 5.3.1 the scenario (similar to the one illustrated in Figure 5.2) where the session is executed in the client machine and interacts with a server remotely. In our calculations, we've already considered the overhead of the message.

We assume the request message size to be of 1 KB. We range the size of the result message from 1 KB to 1000 KB and the time required for the session to complete its task from 1 to 1000s. This comprises the time waiting for the server to process the request

Result size (KB)	message creation		transfer		request processing (sec)			
	request	result	request	result	1	10	100	1000
1	0,046	0,046	0,009	0,008	**1,109**	**10,109**	**100,109**	**1000,109**
10	0,046	0,051	0,009	0,080	**1,186**	**10,186**	**100,186**	**1000,186**
20	0,046	0,057	0,009	0,160	**1,272**	**10,272**	**100,272**	**1000,272**
50	0,046	0,074	0,009	0,400	**1,529**	**10,529**	**100,529**	**1000,529**
100	0,046	0,102	0,009	0,800	**1,957**	**10,957**	**100,957**	**1000,957**
200	0,046	0,160	0,009	1,600	**2,815**	**11,815**	**101,815**	**1001,815**
500	0,046	0,332	0,009	4,000	**5,387**	**14,387**	**104,387**	**1004,387**
1000	0,046	0,619	0,009	8,000	**9,674**	**18,674**	**108,674**	**1008,674**

Table 5.3.1 – Remote interaction time values (1Mbps)

Table 5.3.1 reading:

Message creation indicates the time required creating the request message for the session to interact with the server and the time required to create the result message to return to the client.

Transfer request shows the time elapsed for transferring the request from the client to the server.

Calculation: $\text{Transfer request} = \frac{\text{Request message size}}{Bandwidth\ \boldsymbol{h}}$

Transfer result indicates the time elapsed for transferring the result from the server to the client.

Calculation: $\text{Transfer result} = \frac{Result\ message\ size}{Bandwidth\ \boldsymbol{h}}$

At last, request processing denotes the time spent by the server to process the request.
The total time for each situation is presented in bold.

Calculation:

$$Total\ Time = \text{request creation} + \text{request transfer} + \text{request processing} + \text{result creation} + \text{result transfer}$$

We can observe that the total time (in bold) increases proportional to the result size and to the time of request processing.

5.3.2. Remote execution

In this scenario, we have the same variables as the previous one but this time the session is uploaded to a remote station (similar to the one illustrated in Figure 5.1). The size of the

request message is 30,9326 KB as we considered the migrating session to contain 10 classes (22,645 KB). Like in all calculations, the overhead is already included.

Result size (KB)	MW	message creation		transfer		request processing (sec)			
		request	result	session	result	1	10	100	1000
1	0,214	0,046	0,046	0,247	0,008	**1,561**	**10,561**	**100,561**	**1000,561**
10	0,214	0,046	0,051	0,247	0,080	**1,638**	**10,638**	**100,638**	**1000,638**
20	0,214	0,046	0,057	0,247	0,160	**1,724**	**10,724**	**100,724**	**1000,724**
50	0,214	0,046	0,074	0,247	0,400	**1,981**	**10,981**	**100,981**	**1000,981**
100	0,214	0,046	0,102	0,247	0,800	**2,410**	**11,410**	**101,410**	**1001,410**
200	0,214	0,046	0,160	0,247	1,600	**3,267**	**12,267**	**102,267**	**1002,267**
500	0,214	0,046	0,332	0,247	4,000	**5,840**	**14,840**	**104,840**	**1004,840**
1000	0,214	0,046	0,619	0,247	8,000	**10,127**	**19,127**	**109,127**	**1009,127**

Table 5.3.2 – Remote execution time values (1Mbps)

Table 5.3.2 reading:

In comparison to the previous table, this one has a new column called “MW” which represents the middleware share in the session migration. Another difference is the session transfer column instead of the request transfer column. These changes are all related to the fact that this scenario represents the uploading of a session to a remote station. Thus, the content transferred to the server is not a solo request but a session (therefore, the values in this column are relatively higher).

Middleware (MW) indicates the time spent in constructing the requisites for a session to execute remotely and to return its results to the client.

Calculation:

$$\text{Middleware time} = \text{collect classes} + \text{smash object} + \text{message creation} + \text{build object} + \text{smash result} + \text{transfer result} + \text{build result}$$

Transfer session denotes the time elapsed for transferring the session from the client machine to the server.

Calculation: $\frac{\text{Session size}}{Bandwidth} = \frac{30{,}9326}{125} = 0{,}2474609$

Similarly to the previous scenario, the total time for each situation is presented in bold.

Calculation:

$$Total\ Time = \text{middleware} + \text{request creation} + \text{transfer session} + \text{request processing} + \text{result creation} + \text{result transfer}$$

Although we have a session migrating to a remote station to execute, the total time (in bold) also increases proportional to the result size and to the time of request processing.

5.3.3. Comparison between remote interaction and remote execution

In order to understand the differences between remote execution versus remote interaction, Table 5.3.3 and the respective graph (Fig. 5.3.1) represent the comparison between the previous two tables.

Calculation: $\frac{\text{Results of Table 5.3.1}}{\text{Results of Table 5.3.2}} - 1$

result size (KB)	request processing (sec)			
	1	10	100	1000
1	-28,98%	-4,28%	-0,45%	-0,05%
10	-27,62%	-4,25%	-0,45%	-0,05%
20	-26,24%	-4,22%	-0,45%	-0,05%
50	-22,84%	-4,12%	-0,45%	-0,05%
100	-18,77%	-3,97%	-0,45%	-0,05%
200	-13,85%	-3,69%	-0,44%	-0,05%
500	-7,75%	-3,05%	-0,43%	-0,05%
1000	-4,47%	-2,37%	-0,41%	-0,04%

Table 5.3.3 – Remote interaction versus remote execution (1Mbps)

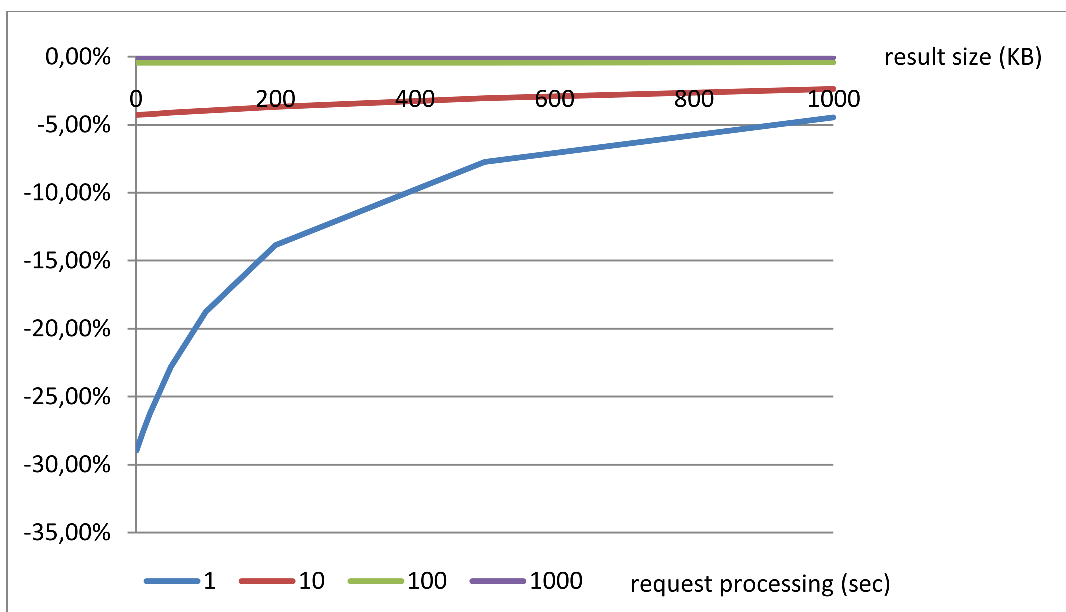


Figure 5.3.1 – Graph of Table 5.3.3

Interpreting the graph of Figure 5.3.1, we are in conditions to conclude that the migration takes always longer than remote interaction in this scenario. The worst case is when the processing time is shorter and/or the result size is smaller. For larger result's size and/or for longer processing time, the difference between migration and remote interaction tends to be smaller.

5.4. Remote interaction versus remote execution (higher bandwidth)

The second evaluation conducted is similar to the previous one but with the bandwidth changed to a higher value: 100 Mbps instead of 1 Mbps. The objective of this test is to understand if the bandwidth can considerably influence the results favoring remote interaction or favoring remote execution.

5.4.1. Comparison between remote interaction and remote execution

Two tables have been constructed similarly to Table 5.3.1 (remote interaction) and Table 5.3.2 (remote execution) but this time with a bandwidth of 100 Mbps. The comparison of these two tables is represented in Table 5.4.1 and the respective graph of Figure 5.4.1.

result size (KB)	request processing (sec)			
	1	10	100	1000
1	-16,57%	-2,10%	-0,22%	-0,02%
10	-16,50%	-2,10%	-0,22%	-0,02%
20	-16,41%	-2,10%	-0,22%	-0,02%
50	-16,18%	-2,10%	-0,22%	-0,02%
100	-15,79%	-2,09%	-0,22%	-0,02%
200	-15,07%	-2,08%	-0,22%	-0,02%
500	-13,26%	-2,04%	-0,22%	-0,02%
1000	-11,05%	-1,98%	-0,21%	-0,02%

Table 5.4.1 –Remote interaction versus remote execution (100Mbps)

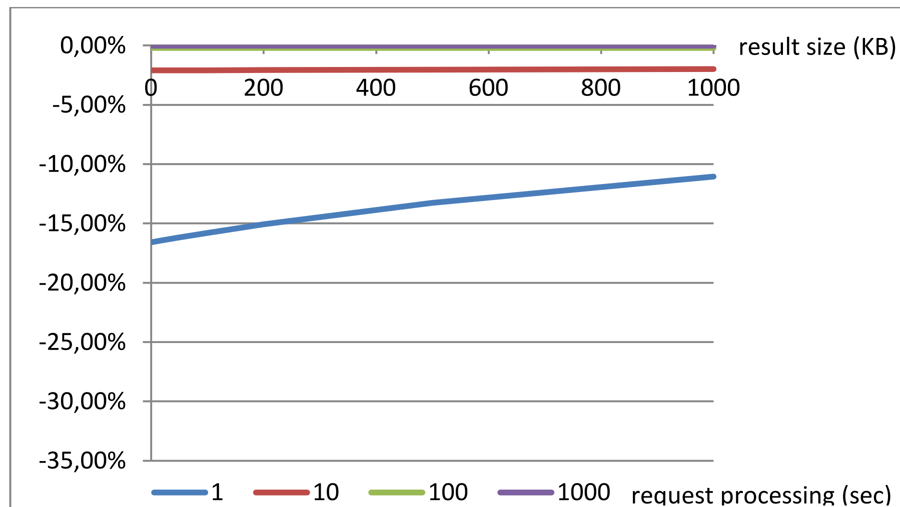


Figure 5.4.1 – Graph of Table 5.4.1

The graph represented in Figure 5.4.1 illustrates that remote interaction is always faster than remote execution in this scenario which provides us the same conclusions as explained in subsection 5.3.3.

Thus, in both scenarios (1 Mbps and 100 Mbps) which featured only one interaction with the server, we can conclude that remote interaction is always worthier than remote execution.

5.5. Interactions

The next evaluation we performed, we used the same base values as the previous one (Section 5.4). We used a fixed result size of 1 KB, set the request processing time to 0 and raised the number of interactions in the server. The objective of this test is to understand the weight of the interactions between a session and a server on remote execution and on remote interaction.

5.5.1. Remote interaction versus remote execution (lower bandwidth)

We start by describing in Table 5.5.1 the scenario where the session is executed in the client machine and interacts with a server remotely. In this test, we range the interactions number from 1 to 60.

Interactions	message creation		transfer		total time
	request	result	request	result	
1	0,041	0,041	0,009	0,009	**0,100**
2	0,081	0,081	0,019	0,019	**0,200**
3	0,122	0,122	0,028	0,028	**0,300**
4	0,162	0,162	0,038	0,038	**0,400**
6	0,243	0,243	0,057	0,057	**0,599**
8	0,324	0,324	0,075	0,075	**0,799**
10	0,405	0,405	0,094	0,094	**0,999**
12	0,486	0,486	0,113	0,113	**1,199**
14	0,567	0,567	0,132	0,132	**1,398**
17	0,689	0,689	0,160	0,160	**1,698**
20	0,810	0,810	0,189	0,189	**1,998**
25	1,013	1,013	0,236	0,236	**2,497**
30	1,215	1,215	0,283	0,283	**2,996**
40	1,620	1,620	0,377	0,377	**3,995**
50	2,026	2,026	0,471	0,471	**4,994**
60	2,431	2,431	0,566	0,566	**5,993**

Table 5.5.1 – Remote interacting with interactions (1Mbps)

Table 5.5.1 reading:

Interactions column denotes the number of interactions between the session and the server. Total time indicates the total time spent by the interactions.

Calculation:

$$Total\ Time = \text{request creation} + \text{request transfer} + \text{result creation} + \text{result transfer}$$

We can easily verify that the total time (in bold) increases proportional to number of interactions.

Next and analogously to the previous table, we present the values (Table 5.5.2) describing the scenario where the session is executed remotely. In this case, we considered the size of the message to be 30,9326KB (including the session).

Interactions	middleware	message creation		Transfer		total time
		request	result	session	result	
1	0,214	0,041	0,041	0,247	0,009	**0,552**
2	0,214	0,081	0,081	0,247	0,009	**0,633**
3	0,214	0,122	0,122	0,247	0,009	**0,714**
4	0,214	0,162	0,162	0,247	0,009	**0,795**
6	0,214	0,243	0,243	0,247	0,009	**0,957**
8	0,214	0,324	0,324	0,247	0,009	**1,120**
10	0,214	0,405	0,405	0,247	0,009	**1,282**
12	0,214	0,486	0,486	0,247	0,009	**1,444**
14	0,214	0,567	0,567	0,247	0,009	**1,606**
17	0,214	0,689	0,689	0,247	0,009	**1,849**
20	0,214	0,810	0,810	0,247	0,009	**2,092**
25	0,214	1,013	1,013	0,247	0,009	**2,497**
30	0,214	1,215	1,215	0,247	0,009	**2,902**
40	0,214	1,620	1,620	0,247	0,009	**3,712**
50	0,214	2,026	2,026	0,247	0,009	**4,522**
60	0,214	2,430	2,430	0,247	0,009	**5,333**

Table 5.5.2 – Remote executing with interactions (1Mbps)

Calculation:

$$Total\ Time = middleware + \text{request creation} + \text{session transfer} + \text{result creation} + \text{result transfer}$$

In this case we can also conclude that the total time (in bold) increases proportional to number of interactions.

Comparison between remote interaction and remote execution

Interactions	Total Time
1	-81,92%
2	-68,46%
3	-58,06%
4	-49,77%
6	-37,41%
8	-28,62%
10	-22,06%
12	-16,97%
14	-12,91%
17	-8,15%
20	-4,50%
25	0,01%
30	3,26%
40	7,63%
50	10,43%
60	12,38%

Table 5.5.3 – Remote interaction versus remote execution with interactions (1Mbps)

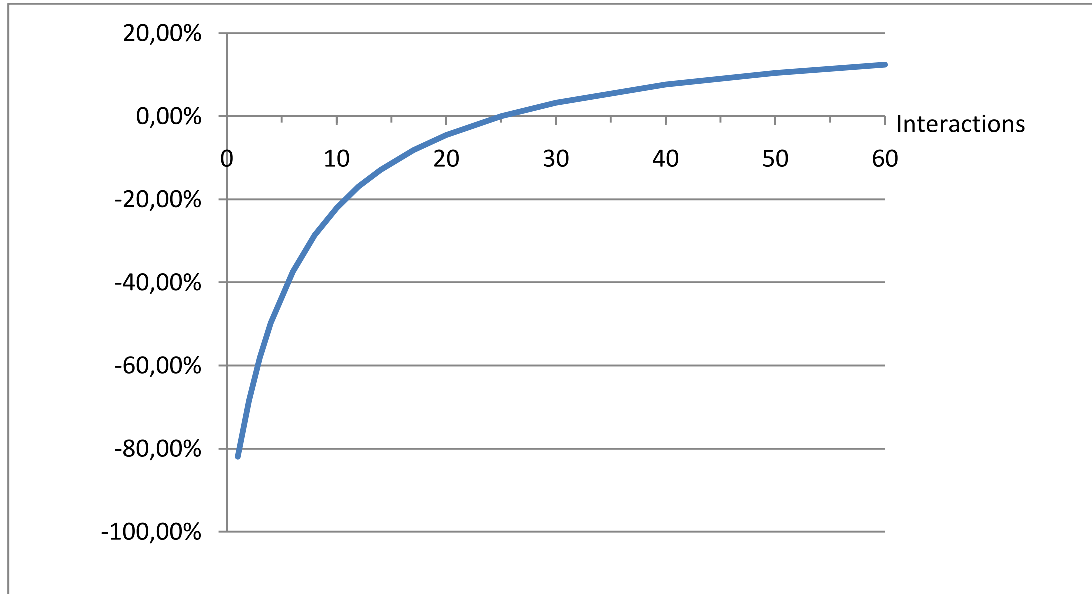


Figure 5.5.1 – Graph of Table 5.5.3

The graph of Fig 5.5.1 allow us to understand that for a result size of 1 KB in a network with a bandwidth of 1 Mbps, a session that requires to interact 25 times or more with a server has better performance if uploaded to the server to interact locally.

5.5.2. Remote interaction versus remote execution (higher bandwidth)

The evaluation performed in this scenario is to understand if the bandwidth readjusted to 100 Mbps has significant impact in the remote interaction versus remote execution performances. In this case, we had to increase the result size from 1 KB to 50 KB in order to see the interaction number in which the remote execution is a 'better solution' rather than remote interaction.

Comparison between remote interaction and remote execution

Interactions	Total Time
1	-71,81%
2	-55,54%
3	-44,96%
4	-37,52%
6	-27,76%
8	-21,64%
10	-17,44%
12	-14,38%
14	-12,06%
17	-9,45%
20	-7,53%
25	-5,26%
30	-3,68%
40	-1,63%
50	-0,36%
60	0,50%

Table 5.5.4 – Remote interaction versus remote execution with interactions (100 Mbps)

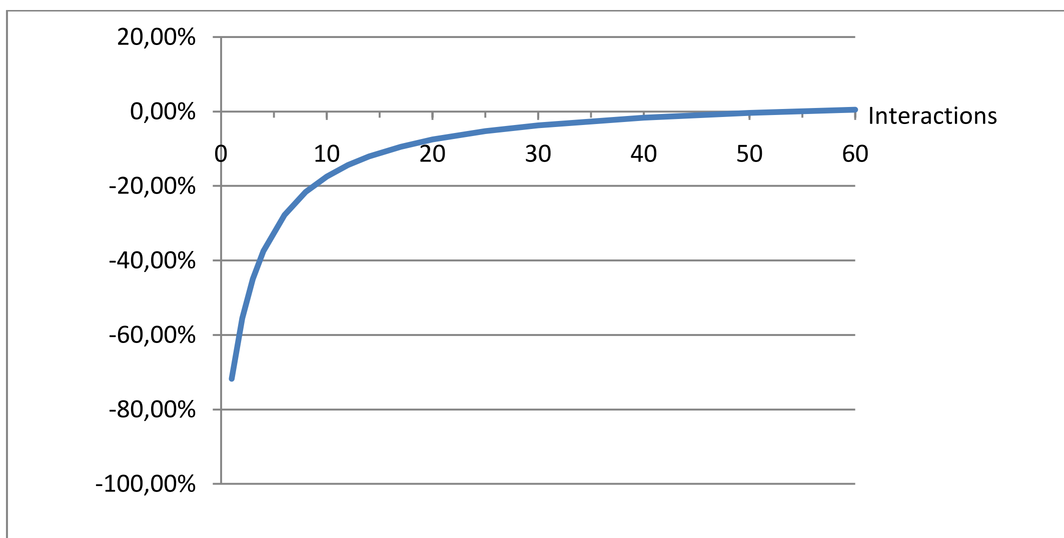


Figure 5.5.2 – Graph of Table 5.5.4

We can conclude that for a result size of 1 KB and a session sized 30 KB, a network with a bandwidth of 1 Mbps offers better performances for remote execution than a network with a bandwidth of 100 Mbps.

5.6. Unidirectional

The next evaluations we realized has traveling strategies included. We begin with the unidirectional traveling strategy. The objective of this test is to understand the weight that remote execution has on a session that is traveling in sequence (unidirectional) from 1 to N servers in comparison to a session that is remote interacting with the same number of servers.

Similarly to previous tests, we performed two evaluations: one with the bandwidth set to 1 Mbps and a second set to 100 Mbps.

In this scenario (Fig. 5.6.1), we had to consider that the time a session spends in the first server is not the same as in the last server or in the servers in the 'middle' (i.e., those that are neither first nor last of the route). This happens because of the 'extra' data that is created due to the computed result. To simplify the scenario, we also considered that the processing time in the server is zero and the size of a session is 30,9326KB (10 classes). This means that when the session leaves the client machine, it contains 10 classes but when it leaves the first server, it is already a little bit larger than the 10 classes. We assumed the 'extra' data to have the size of 3,09326 KB (the size of 1 class). Thus, the first server receives 10 classes, the second (and so on until the last) receives 11 'classes' because of the computed results that have been calculated in the meantime.

The last server sends a component consisting of one class only (result) because the session is not required to travel back to the client. In our calculations, we had to consider these cases in which the classes inputted and outputted to a server vary according to its position in the route used by the session.

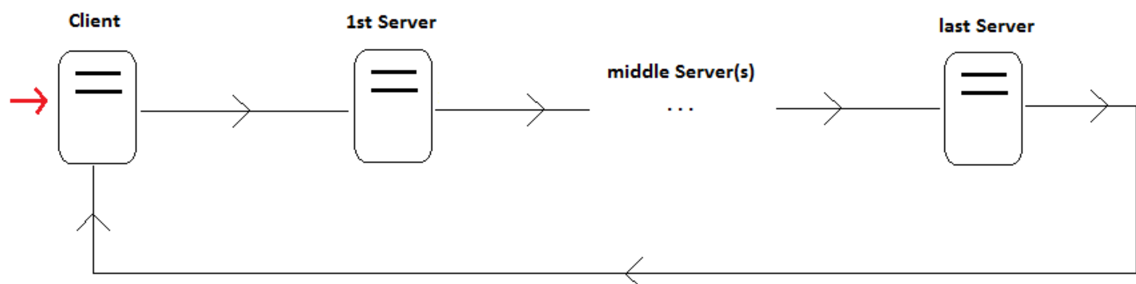


Figure 5.6.1 – Number of classes transferred from host to host

The calculations we used to determinate the values for remote execution (Table 5.6.1 and Table 5.6.4) are as follows:

Total Time =
client side operations(collect classes, message creation, smash object, build object and transfer) +
server side operations(collect classes, message creation, smash object, build object and transfer) +
num servers × (
interactions × (
message creation request + message creation result + request processing))

where:
for all servers:
client side operations = collect classes + session message creation +
session smash object + result build + session transfer

for only 1 server:
server side operations = result message creation + result smash + session build + result transfer

for only 2 servers:
server side operations = 1st server operations + last server operations
where:
1st server operations = session and result message creation + session and result smash +
session build + session and result transfer
last server operations = result message creation + result smash + session and result build +
result transfer)

for 3 or more servers:
server side operations = 1st server operations + middle server operations × (number of servers – 2) +
last server operations
where:
1st server operations and last server operations are the same as for only 2 servers
middle server operations = session and result message creation + session and result smash +
session and result build + session and result transfer)

5.6.1. Remote interaction versus remote execution (lower bandwidth)

Servers	Interactions			
	1	10	100	1000
1	0,586	1,316	8,607	81,525
2	1,048	2,507	17,090	162,926
3	1,511	3,698	25,573	244,327
4	1,973	4,889	34,057	325,729
5	2,435	6,081	42,540	407,130
6	2,897	7,272	51,023	488,531
7	3,359	8,463	59,506	569,932
8	3,821	9,655	67,989	651,333
9	4,283	10,846	76,472	732,734
10	4,746	12,037	84,955	814,135
11	5,208	13,229	93,438	895,536
12	5,670	14,420	101,922	976,938
13	6,132	15,611	110,405	1058,339
14	6,594	16,803	118,888	1139,740
15	7,056	17,994	127,371	1221,141
16	7,518	19,185	135,854	1302,542
17	7,981	20,377	144,337	1383,943
18	8,443	21,568	152,820	1465,344

Table 5.6.1 – Unidirectional remote execution (1Mbps)

For remote interaction (Table 5.6.2), we used the following calculation:

Total Time = number of servers × (
interactions × (message creation request + message creation result + request processing + transfer request + transfer result))

Servers	Interactions			
	1	10	100	1000
1	0,338	3,379	33,791	337,911
2	0,676	6,758	67,582	675,821
3	1,014	10,137	101,373	1013,732
4	1,352	13,516	135,164	1351,643
5	1,690	16,896	168,955	1689,553
6	2,027	20,275	202,746	2027,464
7	2,365	23,654	236,537	2365,374
8	2,703	27,033	270,329	2703,285
9	3,041	30,412	304,120	3041,196
10	3,379	33,791	337,911	3379,106
11	3,717	37,170	371,702	3717,017
12	4,055	40,549	405,493	4054,928
13	4,393	43,928	439,284	4392,838
14	4,731	47,307	473,075	4730,749
15	5,069	50,687	506,866	5068,659
16	5,407	54,066	540,657	5406,570
17	5,744	57,445	574,448	5744,481
18	6,082	60,824	608,239	6082,391

Table 5.6.2 – Unidirectional remote interaction (1Mbps)

Table 5.6.3 and the respective graph in Figure 5.6.2 represent the comparison of the previous two tables (remote execution versus remote interaction).

Servers	Interactions			
	1	10	100	1000
1	-42,37%	156,85%	292,58%	314,49%
2	-35,54%	169,60%	295,44%	314,80%
3	-32,89%	174,12%	296,40%	314,91%
4	-31,48%	176,44%	296,88%	314,96%
5	-30,61%	177,85%	297,17%	314,99%
6	-30,01%	178,80%	297,36%	315,01%
7	-29,58%	179,48%	297,50%	315,03%
8	-29,26%	180,00%	297,61%	315,04%
9	-29,00%	180,40%	297,69%	315,05%
10	-28,79%	180,72%	297,75%	315,05%
11	-28,62%	180,98%	297,80%	315,06%
12	-28,48%	181,20%	297,85%	315,07%
13	-28,36%	181,39%	297,88%	315,07%
14	-28,26%	181,55%	297,92%	315,07%
15	-28,17%	181,69%	297,94%	315,08%
16	-28,09%	181,81%	297,97%	315,08%
17	-28,02%	181,92%	297,99%	315,08%
18	-27,96%	182,01%	298,01%	315,08%

Table 5.6.3 – Unidirectional remote execution versus remote interaction (1Mbps)

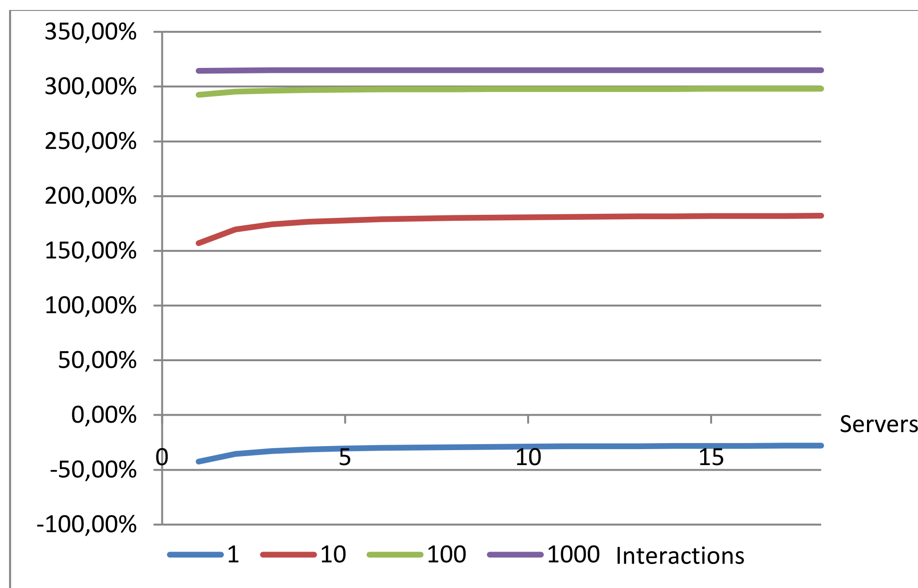


Figure 5.6.2 – Graph of table 5.6.3

Through the graph illustrated in Fig. 5.6.2, we can see that remote interaction has better performances (at least for 18 servers) with one interaction whilst remote execution is better for ten or more interactions.

5.6.2. Remote interaction versus remote execution (higher bandwidth)

This evaluation is equal to the previous one but with a bandwidth of 100 Mbps.

Servers	Interactions			
	1	10	100	1000
1	0,298	1,028	8,319	81,237
2	0,493	1,952	16,535	162,371
3	0,689	2,876	24,752	243,506
4	0,884	3,801	32,968	324,640
5	1,079	4,725	41,184	405,774
6	1,274	5,649	49,400	486,908
7	1,470	6,574	57,616	568,042
8	1,665	7,498	65,833	649,177
9	1,860	8,423	74,049	730,311
10	2,055	9,347	82,265	811,445
11	2,250	10,271	90,481	892,579
12	2,446	11,196	98,697	973,713
13	2,641	12,120	106,914	1054,848
14	2,836	13,045	115,130	1135,982
15	3,031	13,969	123,346	1217,116
16	3,227	14,893	131,562	1298,250
17	3,422	15,818	139,778	1379,384
18	3,617	16,742	147,995	1460,519

Table 5.6.4 – Unidirectional remote execution (100Mbps)

Servers	Interactions			
	1	10	100	1000
1	0,084	0,836	8,359	83,589
2	0,167	1,672	16,718	167,178
3	0,251	2,508	25,077	250,767
4	0,334	3,344	33,436	334,356
5	0,418	4,179	41,794	417,945
6	0,502	5,015	50,153	501,533
7	0,585	5,851	58,512	585,122
8	0,669	6,687	66,871	668,711
9	0,752	7,523	75,230	752,300
10	0,836	8,359	83,589	835,889
11	0,919	9,195	91,948	919,478
12	1,003	10,031	100,307	1003,067
13	1,087	10,867	108,666	1086,656
14	1,170	11,702	117,024	1170,245
15	1,254	12,538	125,383	1253,834
16	1,337	13,374	133,742	1337,423
17	1,421	14,210	142,101	1421,011
18	1,505	15,046	150,460	1504,600

Table 5.6.5 – Unidirectional remote interaction (100Mbps)

Table 5.6.6 represents the comparison between remote execution (Table 5.6.4) and remote interaction (Table 5.6.5).

Servers	Interactions			
	1	10	100	1000
1	-71,99%	-18,66%	0,47%	2,89%
2	-66,12%	-14,35%	1,10%	2,96%
3	-63,59%	-12,81%	1,31%	2,98%
4	-62,17%	-12,03%	1,42%	2,99%
5	-61,27%	-11,55%	1,48%	3,00%
6	-60,64%	-11,22%	1,52%	3,00%
7	-60,18%	-10,99%	1,55%	3,01%
8	-59,83%	-10,82%	1,58%	3,01%
9	-59,55%	-10,68%	1,60%	3,01%
10	-59,33%	-10,57%	1,61%	3,01%
11	-59,14%	-10,48%	1,62%	3,01%
12	-58,99%	-10,41%	1,63%	3,01%
13	-58,85%	-10,34%	1,64%	3,02%
14	-58,74%	-10,29%	1,65%	3,02%
15	-58,64%	-10,24%	1,65%	3,02%
16	-58,55%	-10,20%	1,66%	3,02%
17	-58,47%	-10,16%	1,66%	3,02%
18	-58,40%	-10,13%	1,67%	3,02%

Table 5.6.6 – Unidirectional remote execution versus remote interaction (100Mbps)

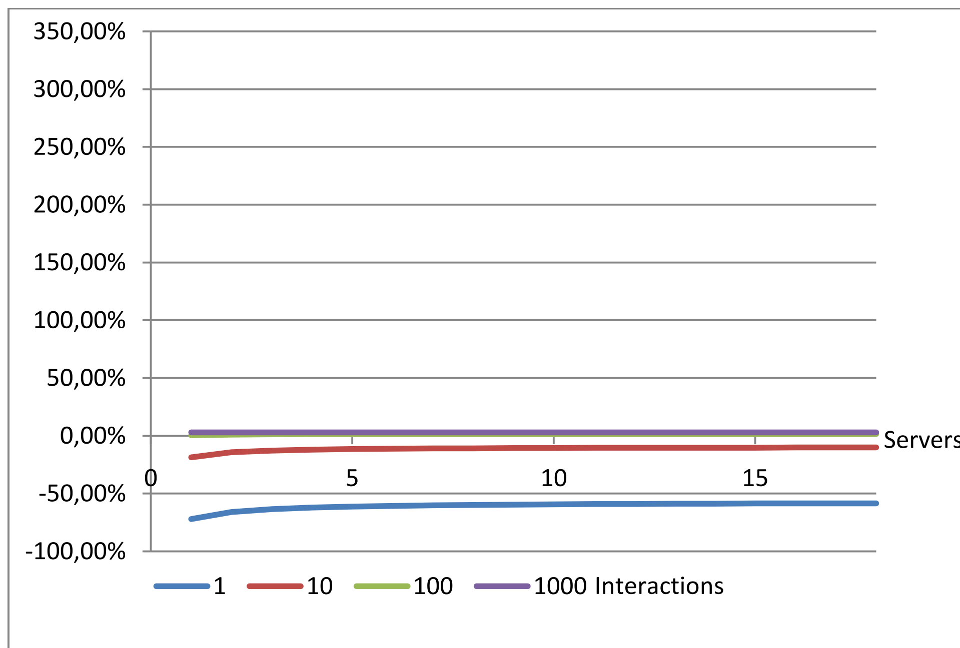


Figure 5.6.3 – Graph of table 5.6.6

The graph described in Figure 5.6.3 allows us to verify that remote execution is better for 100 or more interactions in the server. In addition, and in comparison to the scenario of 1 Mbps bandwidth, we are able to confirm that a higher bandwidth favors remote execution.

5.7. Multidirectional (Speed-Up)

The last evaluation we conducted is to determinate the speed-up while using a multidirectional traveling strategy in a network of 1 Mbps bandwidth and a request processing time of 1 second. The objective is to understand the speed-up gained in sharing the interactions through the targeted servers. Table 5.7.1 specifies the values obtained in our test, in seconds.

Calculation:

Total Time = (collect classes + smash session + message session creation × servers + client build result + client transfer session + server transfer result + server result creation + server smash result + server build session + interactions × (message creation request + message result creation + request processing) / servers)

Servers	Interactions			
	5	20	100	1000
1	1,00	1,00	1,00	1,00
2	1,89	1,97	1,99	2,00
3	2,63	2,90	2,98	3,00
4	3,23	3,77	3,95	3,99
5	3,68	4,58	4,91	4,99
6		5,32	5,85	5,98
7		5,98	6,77	6,98
8		6,58	7,67	7,97
9		7,10	8,54	8,95
10		7,55	9,39	9,94
11		7,94	10,21	10,92
12		8,27	11,00	11,89
13		8,53	11,76	12,86
14		8,75	12,49	13,83
15		8,92	13,19	14,80
16		9,04	13,85	15,76
17		9,13	14,48	16,71
18		9,19	15,08	17,66

Table 5.7.1 – Multidirectional speed-up (1Mbps)

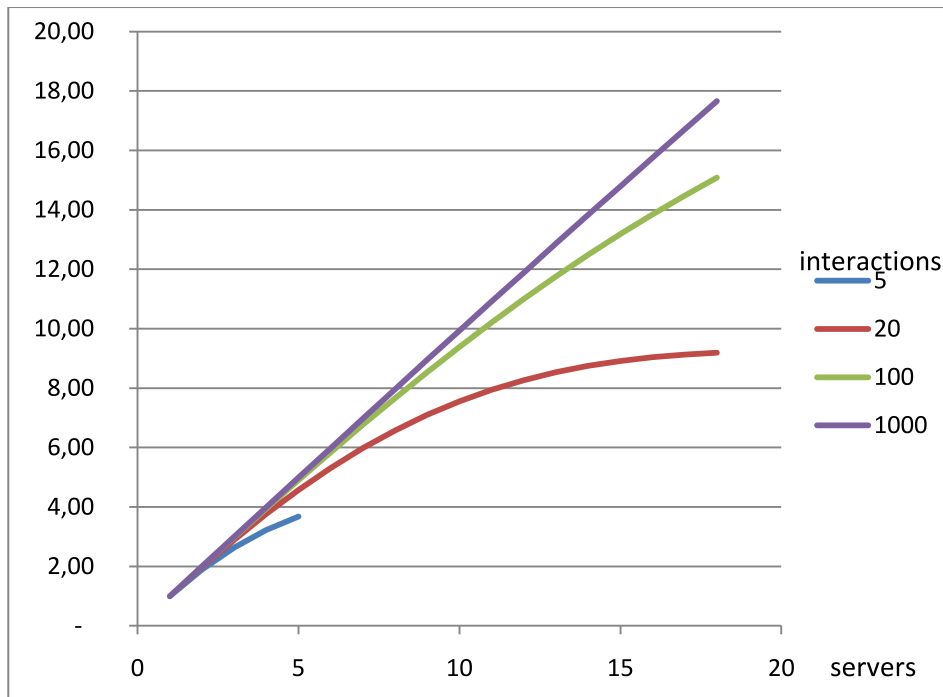


Figure 5.7.1 – Graph of table 5.7.1

The graph represented in Fig. 5.7.1 allows us to trivially understand that as more servers are used in the computation, a speed-up is confirmed. Additionally, the growth becomes linear as the number of interactions is greater.

6. Conclusions

6.1. Summary

The goal of this thesis was to produce a model that comprises of a middleware layer equipped with functionalities that allow user-developed sessions to travel in a network in order to be executed remotely. The middleware is assigned to take control of a session from the moment a user requests to upload it. Tasks such as collecting the classes required to run sessions in remote stations, transferring sessions to the specified hosts, session executing and returning of the computed results back to the client, are all within the responsibility of the middleware. In addition, it is important to mention that the network is available to the client in the form of services, namely Web services, instead of network nodes since we implemented a model whose computing is service-oriented.

The implementation of this model also focused in providing a simple methodology for the programmer to dispatch sessions to the network. It is through an API that the programmer is able to develop components at his/her own taste (the user has total freedom to write the code to run remotely) and to upload them to the network. In addition, this API offers methods that enhance session mobility in the network such as the traveling strategy that it must take. The programmer is also able to bridge a session to another, which means that one session will use the computed results from another session to complete its execution.

The evaluations conducted in the instantiated model allowed us to identify situations that favor session migration and situations that favor remote interaction. An interesting finding is that a faster bandwidth not always benefits session migration.

All objectives proposed for this work were successfully accomplished. We were able to instantiate a model that provides an API for the programmer to develop components to execute in remote stations. Our model offers a middleware layer which is responsible for sessions traveling to any host in the network and to return the computed results to the respective client. At last, we presented evaluations that were performed with the objective of comprehending the middleware impact in session's migration on the model instantiated.

6.2. Future work

Future work on service-oriented mobility consists of embodying the system with security, allowing modification of the itinerary route dynamically at run-time, featuring a “go home” method to instruct the itinerary that it is time to return to the client machine, implement integration with Web service registries and implement integration with service level agreements.

7. Bibliography

[1] Luc Clement and Andrew Hately and Claus von Riegen and Tony Rogers - UDDI spec technical committee draft 3.0.2. Oasis committee draft, 2004.

[2] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems – Towards the Programmable Internet, pages 111–130. Springer-Verlag, 1997.

[3] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Hen- rik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (SOAP) 1.1. W3C Note NOTE-SOAP-20000508, World Wide Web Consortium, May 2000.

[4] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. W3C, 1.1 edition, March 2001. http://www.w3c.org/TR/wsdl.

[5] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. IEEE Trans. Softw. Eng., 24(5), 1998.

[6] Graham Glass. Overview of Voyager: ObjectSpace's Product Family for State-of-the-art Distributed Computing. Technical report, CTO ObjectSpace, 1999.

[7] Mehdi Jazayeri and Wolfgang Lugmayr. Gypsy: A Component-based Mobile Agent Sys-tem. In Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000), 2000.

[8] Gunter Karjoth, Danny B. Lange, and Mitsuru Oshima. A Security Model for Aglets. IEEE Internet Computing, 1(4), 1997.

[9] Danny B. Lange. Mobile Objects and Mobile Agents: The Future of Distributed Computing? In Proceedings of The European Conference on Object-Oriented Programming '98, pages 1–12, 1998.

[10] Sun Microsystems Qusay H. Mahmoud. Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI). http://java.sun.com/developer/technicalArticles/WebServices/soa/index.html, 2005.

[11] Pedro Marques, Sergi Robles, Jordi Cucurull-Juan, Ricardo Correia, Guillermo Navarro, and Ramon Martí. Secure integration of distributed medical data using mobile agents. IEEE Intelligent Systems, 21(6):47–54, 2006.

[12] Sun Microsystems. Introduction to CORBA. http://java.sun.com/developer/onlineTraining/corba/corba.html, 1999.

[13] Dejan S. Milojicic, Frederick Douglis, and Richard G. Wheeler, editors. Mobile Agents for Mobile Computing. Addison Wesley and ACM Press, April 1999.

[14] OASIS. Web Services Notification (WSN) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn.

[15] OASIS. Web Services Security (WSS) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

[16] Object Management Group. The Common Object Request Broker: Architecture and Specification. Object Management Group, 2001.

[17] Ed Ort. Service-Oriented Architecture and Web Services: Concepts, Technologies, and Tools. http://java.sun.com/developer/technicalArticles/WebServices/soa2/soa2.pdf, 2005.

[18] Hervé Paulino. An Overview of Mobile Agents Systems. Technical report, CITI, 01 2002.

[19] Hervé Paulino. An Abstract Machine for Service-oriented Mobility, volume 2 of Chapman & Hall/CRC Computational Science, pages 199–233. CRC Press, William Gardner and Michael Alexander edition, 12 2008.

[20] Hervé Paulino. A Service-Oriented Approach to Software Mobility. Submitted, 2009.

[21] Hervé Paulino and Luís Lopes. A programming language for service-oriented computing with mobile agents. Software: Practice and Experience, 38(7):705–734, 06 2008.

[22] Hervé Paulino and Carlos Tavares. Sedeuse: A model for service-oriented computing in dynamic environments. In Carlo; Magedanz Thomas Bonnin, Jean-Marie; Giannelli, editor, Mobile Wireless Middleware, Operating Systems and Applications. Second Inter- national Conference, Mobilware 2009, Berlin, Germany, April 28-29, 2009, number 7 in Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommu- nications Engineering, pages 157–170. Springer-Verlag, 04 2009.

[23] Sun Microsystems. An Overview of RMI Applications. http://java.sun.com/docs/books/tutorial/rmi/overview.html.

[24] Sun Microsystems, Inc. JAX-WS Reference Implementation Project. https://jax-ws.dev.java.net.

[25] The Apache Software Foundation. Apache Axis2. http://ws.apache.org/axis2.

[26] Victoria Shannon. A 'more revolutionary' Web. New York Times, 2006. http://www.nytimes.com/2006/05/23/technology/23iht-Web.html.

[27] James E. White. Telescript Technology: Scenes from the Electronic Marketplace. General Magic White Paper, General Magic edition, 1995

[28] Jansen, W., Karygiannis, T.: Mobile Agent Security. Special Publication 800-19, National Institute of Standards and Technology (1999)

[29] Brooks, R.R.: Paradigms and security issues. IEEE Internet Computing 8(3) (2004) 54-59

[30] Bettini, L. and Nicola, R. D. 2002. Translating Strong Mobility into Weak Mobility. In Proceedings of the 5th international Conference on Mobile Agents (December 02 – 04, 2001). G. P. Picco, Ed. Lecture Notes In Computer Science, vol. 2240. Springer-Verlag, London, 182-197.

[31] Bettini, L., De Nicola, R., and Pugliese, R. 2002. KLAVA: a Java package for distributed and mobile applications. Softw. Pract. Exper. 32, 14 (Nov. 2002), 1365-1394. DOI= http://dx.doi.org/10.1002/spe.486

[32] Arjav J. Chakravarti, Xiaojin Wang, Jason O. Hallstrom, Gerald Baumgartner, "Implementation of Strong Mobility for Multi-Threaded Agents in Java," Parallel Processing, International Conference on, p. 321, 2003 International Conference on Parallel Processing (ICPP'03), 2003

[33] Distributed Component Object Model (DCOM) http://download.microsoft.com/download/a/e/6/ae6e4142-aa58-45c6-8dcf-a657e5900cd3/[MS-DCOM].pdf

[34] Enterprise JavaBeans Technology http://java.sun.com/products/ejb/index.jsp

[35] Introduction To Enterprise Java Bean(EJB) http://www.roseindia.net/javabeans/javabeans.shtml