

Interactive Distributed Computing Middleware

Ziad Riad Rahhal

A thesis submitted for the degree of
Master of Science in Grid Computing

Section Computational Science
Universiteit van Amsterdam



Supervised by Dr. Robert Belleman

May 21, 2007

To my loving family

And to the memory of my uncle

Rabih Elias Rahhal

Acknowledgments

First, I would like to thank my supervisor *Robert Belleman* for offering this challenging and cutting-edge research topic and for all the support he provided for this research to be successful. Robert is technically skillful as much as he is a senior researcher, so I had the chance to learn both aspects from him. He always gave me from his time and I always ended up learning new things from our conversations.

Thanks to *Bram Stolk* for his smart programming touches on LinuxTuples and for answering questions. Bram; by the time I finished this thesis, you left SARA and moved to Vancouver. I guess the people at SARA will have the hardest time ever to find someone with your skills to fill this position.

By the time I was writing this thesis, I shared the same computer-lab with *Joost Geerdink*. Thanks Joost for being simple and different. Your company during my last months at UvA made my working time much easier. Do you think we will get another chance to spend queen's day on a boat again?

I always enjoyed talking to *Jina Menke* about everything. We share a lot of the same thoughts. Jina, you provided help in different occasions and you were always supportive. Are you still planning to live and work in the Caribbean?

I know he doesn't like it to thank him because he doesn't consider it as if he is making a favour, but I want to thank him anyway; *Ali Hamieh*, thanks for all the support you offered during my stay in Amsterdam. You always helped me during the difficult periods. I also owe Ali's sisters; *Nahed* and *Iman*, and his brother in law *Hussein Daher* that live in France a great deal of thanks for generously hosting me when I was homeless for fifteen days in Amsterdam.

Of course thanks to my family members that encouraged me and offered great love and care. Uncle *George*, you are always there for everyone. The experience you have in life, business and science makes your advices precious and to be remembered. I keep trying to convince *Nayla* to mention somewhere that I am the first person in Lebanon to have assembled your new genius design of transducers. But she is never convinced.

Thank you *Ayham* for your kindness and your sincere personality that I like a lot. *Elissar*, I consider you my second mother. You always took care of me and offered unlimited support. Thanks for everything.

Rabih, I don't know if I call you uncle or father, but I know how much you are happy for me wherever you are right now. I simply miss you a lot. I will take this opportunity to thank your wife *Elham* and my cousins *Dima*, *Nadim*, *Lama* and *Rawad* for their unconditional support.

Tante *Mimi*, I wonder who helped in carrying the thousands of bags that you brought with you this year? I smile everytime I remember you.

My idol father *Riad*, I noticed during my stay away from home that most of the things you told me about especially life, people and politics were right. I saw them. I missed those conversations. No words can express how much I am proud to be your son.

Mama you are simply the best mother on earth. I keep trying to count how many kids you raised, but I never managed. I am fanatic about you; I always think that if all women can raise kids your way and as much as you did, this world would become a better place to live in. You need to teach me how to cook again, I will take it seriously this time because I miss your delicious food.

Marwan, I don't know where to start from. I always relied on you my brother and you were always there to back me up. You made a lot of things easier for me. Thanks for all the things that I couldn't describe here. I would like to thank your wife *Rana* as well for her support and for her kindness that is rare to find. My nephews *Riad* and *Emile*, you are my motivation to continue during the difficult periods.

Nayla, it wouldn't have been possible for me to go abroad and study without your support. This is just one of many supports you offered to me since I was a kid. I cannot find the words to express how important you are for every single person in this family. Wherever you studied or worked, your brilliance has always risen the bar so high that made it the criterion to which the rest is compared. You are the best sister. Thanks for everything.

Abstract

Interactive distributed systems are complex software systems that consist of several (2 or more) computational components that execute on geographically distributed locations at the same time, with one of these components under the direct control of a human user. Examples of interactive distributed systems are computational steering environments, human-in-the-loop simulations, collaborative environments, conferencing tools, shared virtual reality environments, telepresence environments and networked games.

An *Interactive Distributed Simulation and Visualization* (IDSV) environment is a special case of an interactive distributed system. An IDSV consists of an iterative simulation component that is explored by a human through a visualization component. This thesis focuses on issues related to the distributed nature of the IDSV environment. IDSVs are discussed in addition to their requirements. Suitable communication architectures for this type of applications are presented. LinuxTuples, an implementation of the Linda coordination model is selected among these architectures to be put under test in this thesis. LinuxTuples is introduced, tested and used to implement two IDSV prototype test cases. These prototypes challenge different aspects in LinuxTuples and allow the identification of its features and its weakness in IDSV applications.

IDSVs are often used by multiple groups of scientists in collaborative problem solving environments across geographical areas. These groups come together to share their knowledge as well as their computational technologies (software and hardware). Web services are one of these technologies that are widely used and increasingly gain interest in the Web and distributed computing communities. For that reason, a bridge model is studied in this thesis to integrate Web services and Linda systems.

Contents

1	Interactive Distributed Simulation and Visualization	1
1.1	Introduction	1
1.2	Structure of an IDSV environment	2
1.3	Components in an IDSV	3
1.3.1	Iterative simulation	3
1.3.2	Scientific visualization	3
1.3.3	Renderer	3
1.3.4	Interactive exploration	4
1.4	A distributed environment	4
1.5	Outline of this thesis	5
2	Communication and Coordination Architectures	7
2.1	Introduction	7
2.2	Requirement analysis	7
2.2.1	End-users requirements	8
2.2.2	Functional requirements	8
2.2.3	Performance requirements	9
2.2.4	Software engineering requirements	10
2.2.5	Operational requirements	11
2.3	IDSV supporting architectures	11
2.3.1	Point-to-point coordination using CAVERN	11
2.3.2	Distributed Interactive Simulation	11
2.3.3	Linda coordination and communication model	12
2.3.4	Web services	15
2.4	Discussion	16
3	LinuxTuples: Characteristics and Evaluation	19
3.1	Introduction	19
3.2	LinuxTuples	19
3.2.1	Three basic elements	19
3.2.2	Operations	20
3.2.3	Matching mechanism	21
3.3	Examples	21
3.3.1	Add/Subtract functions	21
3.3.2	An average function	23
3.3.3	A frequency function	24
3.4	Performance analysis	25
3.4.1	Roundtrip time	25

CONTENTS

3.4.2	Throughput	30
3.5	Conclusion	31
4	Integrating Web services and Linda	33
4.1	Introduction	33
4.2	An introduction to Web services	33
4.3	Differences between Web services and Linda	35
4.4	A bridge between Linda and Web services	35
4.4.1	Approach	35
4.4.2	Requirements	36
4.4.3	The SOAPpy toolkit	38
4.4.4	Implementation	38
4.4.5	Results	39
4.4.6	Discussion	40
4.5	Conclusion	42
5	Prototype Use Cases	43
5.1	Introduction	43
5.2	Simulated vascular reconstruction	43
5.2.1	Structure of the application	44
5.2.2	Implementation	44
5.2.3	Results	45
5.2.4	Discussion	47
5.3	Collaborative rigid body dynamics	48
5.3.1	Structure of the application	48
5.3.2	Implementation	48
5.3.3	Results	50
5.3.4	Discussion	51
5.4	Conclusion	51
6	Summary and Future Directions	53
6.1	Summary	53
6.2	Conclusion	54
6.3	Future work	55

Chapter 1

Interactive Distributed Simulation and Visualization

1.1 Introduction

Imagine a flight school or an airline company using real airplanes to train their pilots on how to deal with emergency situations and different kinds of technical problems they might face in everyday's flights. This is extremely dangerous, life threatening and highly expensive. The alternative is a computer flight simulation that provides the same environment as in a cockpit with all the required settings to simulate real flights. The result is a safe system that is extendable to more settings, new types of airplanes and that is cost effective. Computer simulations are not restricted to that area though. They are in fact having a widespread use in different scientific fields. This is due to the significant advance that computing power has been witnessing (moore's law). Scientists are now able to explore new areas through highly complex computer simulations in order to understand complex phenomena in science, medicine, financial markets, social science and many other complex systems.

Many computer simulations belong to the same class and share the following four characteristics: (1) they are controlled by a set of parameters, each of which result in a different simulation outcome and these parameters are not defined *a priori*. (2) they are intractable. That is, their end-result can be obtained only through explicit execution from start to end without short cuts and each execution can take hours or days to complete. It is time consuming to repeat these long executions with different sets of parameters in order to reach satisfactory results. The alternative is human intervention where end-users tweak parameter values at run-time and change the direction of the simulation, thus leading it toward satisfactory results without any need for complete repetitions. This process is known under different terminologies like: human in the loop simulation, computational steering, interactive exploration, etc. In this thesis, it is called *interactive simulation*. (3) this class of simulations generate large and complex data volumes that need human analysis and exploration to obtain knowledge and full insight in the end-result. Inspecting such results numerically is hard to achieve, because designing a suitable numerical algorithm requires a prior knowledge of the underlying data structure and that is not always possible. The alternative is to convert the data into a visual representation that offers humans more insight. Interaction with the visualization by changing its parameters offers different representations of the underlying result using different visualization techniques. Moreover, it is often the case where the data is three or four dimensional. For that reason, changing rendering parameters offers better and different views of simulation results. (4) these simulations are NP-complete. That means for an increase in problem size, the computing time and/or memory requirements of the simulation will grow exponentially. This implies increase demands on visualization and rendering processes as well. Therefore, running every

component (simulation, visualization and renderer) on specialized hardware may increase the execution performance and reduce computing time. In many cases the specialized hardware reside on distributed systems over geographic areas and belong to different institutions. As a consequence, components must be geographically distributed in order for each to run on a specialized machine.

1.2 Structure of an IDSV environment

Figure 1.1 presents the structure for an interactive distributed simulation and visualization (IDSV) environment. It consists of four components: (1) the simulation, (2) the visualization, (3) the renderer and (4) the interaction that is directly under a human’s control.

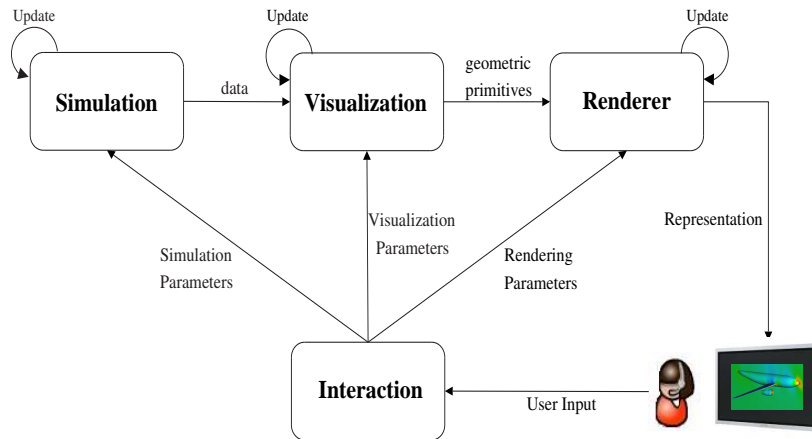


Figure 1.1: *Schematic representation of an Interactive Distributed Simulation and Visualization environment (IDSV). Based on [1].*

The simulation is an iterative process that repeatedly generates data. The visualization maps raw simulation data into geometric primitives that can be understood and rendered by the renderer component using computer graphics. Results are displayed on various kinds of visualization devices to give users better insight in the underlying data set with the intention of deriving knowledge. At the interaction level, the analyst is provided with a user interface to modify simulation parameters and change the course of the simulation at run-time. Instead of running it until completion for each set of parameter values, the user has the ability to monitor intermediate simulation results, while the simulation is running. This allows the simulation to be interrupted if the user notices that it is going to the wrong direction. Mechanisms are also provided to change visualization parameters and apply different visualization techniques. This offers the ability for the user to change the representation of simulation data. Interaction with the renderer is also possible. The user can change rendering parameters (i.e. camera projection) of the visual representation to obtain different and better views of the data. This type of interactions allows analysts and researchers to explore data spaces and/or parameter spaces interactively in order to gain insight in the underlying problem.

1.3 Components in an IDSV

The distribution style of an IDSV, in addition to its four components (simulation, visualization, renderer and interaction), all together give the environment a multidisciplinary character. The different components in such an environment are explained throughout this chapter to briefly identify some of the challenges that could be faced in each case.

1.3.1 Iterative simulation

Computer power doubles every 18 months (Moore's Law) while storage capacity is doubling every 12 months demonstrating a dramatic growth in online data. This significant advance in computing is giving scientists the opportunity to explore new areas through highly complex computer simulations.

Many of these simulations are suitable to be explored using IDSV environments. Examples include: *flow simulations* that are multidimensional (often ≥ 3) and time dependent. They involve large numbers of variables each with multi-dimensional allowed freedom that spans parameter spaces. These simulations are computationally expensive and require parallel architectures to reduce their computational time. The *Lattice Boltzmann* technique is one of the methods used for solving these simulations. Flow simulations are suitable for interactions because they are iterative. Through an IDSV environment scientists can monitor the iterative intermediate results and predict where the simulation is going. Another example is *weather forecasting simulations* that generate large data volumes. These require massive computing power to solve complex equations that describe the atmosphere. In addition, they use huge amounts of collected data for high resolution weather forecasts.

1.3.2 Scientific visualization

Complex and large data volumes generated by the type of simulations discussed in the previous section must be analyzed or explored to get full insight in their patterns. Inspecting such complex results numerically is hard to achieve because it is not always easy to design adequate numerical algorithms due to the complexity of the underlying data. The alternative is to convert the data into a visual representation that allows for human exploration. This approach is often considered the only alternative since the human visual system is known to be the most important of the human senses. The eye has a high bandwidth input to the brain and the visual system can recognize complex structures. However, visualization in science can be dangerous; it could either exclude data patterns or put more emphasis on particular patterns leading to a subjective interpretation of the representation and thus, impeding objective science.

The representation of data that originates from scientific applications is called scientific visualization. It is the process of mapping raw data into geometric primitives (i.e. points, lines, triangles, etc.) that are rendered using computer graphics.

Visualization and rendering are sometimes decoupled into different components. When data sets are of the volume and complexity of those generated by simulations described in the previous section, the process of visual representation becomes computationally expensive and requires specialized resources. The fact that graphics hardware (i.e. graphics cards) are doubling in performance every nine months raises the motivation to decouple and distribute the visualization and the rendering processes on separate dedicated devices. This helps providing users with efficient representations of data for visual analysis and comprehensive exploration in IDSV environments.

1.3.3 Renderer

The visualization component extracts a model out of raw simulation data and the renderer generates the image using computer graphics. Attributes that are rendered include geometry, viewpoint, texture

and lighting. Interaction with the renderer component allows end-users to modify parameters like, for example, camera projection and lighting.

The performance of the renderer depends on the underlying hardware, specifically the graphics card. Some of these are dedicated to specific types of visualization techniques. For example, volume rendering is more computationally expensive than surface rendering, thus volume rendering performs better on graphics cards that are specially designed for it. Decoupling the visualization from the renderer allows low performance devices to use high performance graphics resources of another system. In addition, this allows collaborative visualization by communicating the image to multiple display devices.

1.3.4 Interactive exploration

Interaction in an IDSV environment involves a user changing parameter values of a “live” simulation and of a visualization or of a renderer component.

With interactive simulations, experts can monitor intermediate iterative results and predict the course of the simulation. If the expert notices that the simulation is not converging toward the optimum solution, he/she can interact with the simulation by changing its parameters. Each interaction with the simulation changes its direction and results in new information presented to the user. This speeds up the process of reaching satisfactory results (i.e. reaching the optimum) without the need to repeat computational and time expensive simulations for new sets of parameter values.

Interactive visualization offers the ability to apply different visualization techniques and obtain different representations of the underlying data set. Interaction with rendering parameters (i.e. light, projection, viewpoint) allows different and better views of the visual representation. For example, changing the viewpoint parameter is important for three dimensional data sets. It offers the ability to explore the 3D visual representation from different points.

Interaction in a successful IDSV environment is non-deterministic. This means that the environment will have no prior knowledge of when the user is going to perform an interaction and with which component that interaction is going to take place. This has implications for the design of an IDSV. In addition, interaction implies qualitative characteristics. The work in [2] argues that an IDSV must address the following:

- **Responsive interaction.** Delays will always occur between the moment a user interacts with a component and the moment feedback is received. This can be caused by network latency and/or reduced availability in computational resources. To ensure responsive experience with the environment as a whole, this delay must be minimized.
- **Interaction mechanisms.** Allow the user to change parameters of a live simulation and of a visualization or of a renderer.
- **Intuitive interaction.** Through a rich set of interaction methods that allows users to modify the parameters that control the representation and get different views of the result. A careful design is needed to create an intuitive interface in order to provide a user friendly experience.

1.4 A distributed environment

Specialized hardware and system architectures are nowadays becoming more available. For instance, parallel computing platforms and vector processors are available for executing parallelized or vectorized code. Graphics Processing Units (GPU) provide a performance increase for rendering complex 3D scenes. The speed of communication networks connecting different computational resources is doubling every nine months. With all this computational and network availability, it may be advantageous to distribute the components of an IDSV system over dedicated geographically dispersed hosts. This

reduces the execution time of individual components which may reduce the execution time of the system as a whole. The result of this may contribute to a reduction in financial costs by sharing expensive resources between several organizations. The development and the maintenance of the system becomes easier and decoupled as well.

Beside its advantages, distributed computing is not all bright and problem free. All kinds of complications and challenges are to be expected when dealing with a distributed environment especially in the case of a distributed and dynamic interactive system. The different issues related to the distributed nature of the IDSV environment are the focus of this thesis.

1.5 Outline of this thesis

All components of an IDSV environment need to collectively work in order to solve highly complex and unpredictable problems. The research in this thesis focuses on finding a suitable architecture on top of which an IDSV application can be built. Such an architecture must provide a reliable, flexible and generic means of communication and coordination between multiple and different types of components. It must successfully offer solutions to classical problems found in distributed computing systems. Moreover, and more importantly, it is required to deal with challenges and issues that result from the dynamic character of IDSVs.

Another important aspect of the architecture is that it must be generic and supports components developed with advanced and widely used technologies. From that perspective, this thesis also focuses on integrating the architecture with different highly adopted technologies like Web services.

Chapter 2 discusses requirements that an IDSV environment imposes on the underlying communication architecture. Several architectures are introduced and compared, and the most suitable for IDSVs is chosen. Chapter 3 explains and tests the selected architecture in more detail. A bridge model to integrate Web services and the selected architecture is studied in chapter 4. Chapter 5 presents several implementations of prototype use cases. The thesis ends in chapter 6 with discussions, conclusions and future work.

Chapter 2

Communication and Coordination Architectures

2.1 Introduction

The performance of modern computer architectures allows all components of an IDSV environment to be hosted on the same system. However, running each component on a specialized resource may increase the performance of the system as a whole. For instance, dedicated system architectures are more suitable to execute complex simulations. Special purpose graphic cards are capable of rendering complex 3D scenes. These resources often belong to different geographically distributed organizations and therefore, IDSV components must be distributed in order to use them. The available network speed that guarantees high bandwidth connections between distributed resources supports this distributed model. In addition to performance increase, distribution allows different groups to share their resources and collaborate. However, this distributed model introduces several complicating factors due to traditional issues faced in distributed systems and due to the dynamic and multidisciplinary character of the underlying environment.

The system is meant to be used in highly demanding applications, where real-time responsiveness is required in order to provide a useful and comprehensive environment. In that sense, an IDSV needs to fulfill a number of requirements for it to be successful. These are analyzed in this chapter from several perspectives and a number of existing solutions for the construction of IDSVs are presented and studied.

2.2 Requirement analysis

IDSV applications impose requirements that a communication architecture must meet in order to be suitable for this type of applications. These requirements are discussed from different perspectives that make up a successful IDSV. Communication architectures are later studied and compared against the following five requirements:

1. **End-user.** The human is the center of the universe in any IDSV application. It is that user who is going to perceive results, interact with components and data sets, and acquire knowledge. Therefore, end-user's requirements are important. From these requirements subsequent ones can be derived.
2. **Functionality.** This specifies what functions a communication and coordination architecture is required to offer in order to successfully support IDSV applications and in consequence, support

end-users' needs.

3. **Performance.** Responsiveness is crucial for an efficient, comprehensive and interactive environment. For that reason, the underlying architecture must ensure real-time feedback to end-users' interactions.
4. **Software engineering.** IDSV applications must be relatively easy and flexible to engineer. This ensures software reuse, time saving and scalable applications.
5. **Operational.** Mechanisms must be provided to easily execute and manage IDSV applications.

These different perspectives are discussed in more detail in the following sections:

2.2.1 End-users requirements

- **Reliability.** Errors must have minimal but non fatal effects on the execution of the overall system. Moreover, the environment must provide correct and comprehensive information to end-users. For example, in visualization; information clutter must be avoided and high frame rates must be ensured in order to provide meaningful results.
- **Responsiveness.** Users must receive results almost instantaneously without major delays. Any delay results in a frustrating user interaction and an environment that is awkward to use. In virtual environments this might causes cybersickness symptoms like nausea, eye strain and dizziness.
- **Flexibility.** Especially in terms of the interaction abilities. Users interactions must not be defined *a priori*. These interactions are non-deterministic. That means it is not known when an interaction takes place and with which component. Moreover, the number of components must not be predefined. New components must be able to join and leave at run-time to ensure a broader usage of the system in terms of different configurations.
- **Collaboration.** The ability to share the environment among multiple end-users to work together on common problems. This requirement is related to the possibility of new components to join/leave at run-time.
- **Ease-of-use.** The environment must be easy to use and the end-user must be able to fully explore its functionality. A rich but simple to use interface is required for that purpose and also to shield end-users from the complexity if the architecture.

2.2.2 Functional requirements

- **Coordination and communication.** Components of an IDSV environment exchange data and coordinate their tasks as well. In addition, end-users interact with each component by modifying parameter values. In consequence, an architecture is required to provide a mean of coordination and communication between the components of the system.
- **Distribution.** Components of an IDSV environment are not only spatially distributed over multiple resources, but must be distributed in "time" as well. That is, components are not required to be up and running at the same time in order to communicate. This allows components to join/leave the environment asynchronously depending on the demands of the application. Such a temporal distribution functionality must be ensured in a way that does not interrupt the overall execution of the system.

- **Persistency.** An IDSV environment must have the ability to maintain state and provenance of the history of system interactions. This requirement is linked to the previously mentioned one for temporal distribution. Through persistency, temporally distributed components are able to communicate with other temporally distributed components.
- **Lock-step and pipelined execution.** An execution strategy influences the responsiveness of the environment and its usefulness for users. The choice of a suitable strategy depends on the kind of components (i.e. their update rate) involved in the system. In a “lock-step” execution strategy the simulation is directly controlled by the user. While the user is exploring results from a given time-step, the simulation and the visualization wait for the user’s signal that would make them continue. This strategy is adequate in the case where the update time of each component is negligible, because the user receives interaction results in short notice. On the other hand, if these update times are long, the user must wait for data of the next step to become available and then rendered before receiving the result. This is a situation that is not favorable in interactive systems. The alternative is a “pipelined” execution strategy where components (mainly the simulation and the visualization) are free to advance without any user control. In this case, the user at the interaction level benefits from already available results produced by the simulation.
- **Time management.** Because of execution and communication delays and because of the non-deterministic amount of time an end-user spends on interaction, a situation may occur where each component is processing a different time-step at the same wall-clock time. The user ends up interacting with old data computed by the simulation in the past. The consequence is an inconsistent environment. The solution to this time causality violation is *time management* that deals with the exchange of time stamped information between components [3]. With time management, components inform each others of steps they are currently processing.

2.2.3 Performance requirements

- **Update rate.** Components of an IDSV do not only execute and produce results, they communicate their results to other components in the system and they exchange parameter values. This causes communication delay in addition to the execution time of each component. The update time or T_U of the system is the sum of the execution time of the different components (T_{sim} for the simulation, T_{vis} for the visualization and T_{ren} for rendering) and the communication delay between components ($T_{sim \rightarrow vis}$ and $T_{vis \rightarrow ren}$):

$$T_U = T_{sim} + T_{sim \rightarrow vis} + T_{vis} + T_{vis \rightarrow ren} + T_{ren} \quad (2.1)$$

The update frequency f_U is equal to $\frac{1}{T_U}$. In typical virtual reality environments f_U must be higher than 20Hz in order to provide an immersive and responsive experience for end-users.

- **Response time.** The response time or T_R is the delay between the moment an interaction takes place until the moment the system has reacted to this interaction. Response time depends on the component to which the interaction is directed. For example, when the end-user interacts with the simulation, $T_R = T_i(sim) + T_U$, where $T_i(sim)$ is the delay between the moment the interaction was initiated and the moment it is received by the simulation [1].
To enhance the responsiveness and the performance of the system we need to reduce the execution time of each component (i.e. through dedicated architectures and optimized algorithms) and decrease transfer time of large data volumes between components.
- **Latency.** Delays like $T_i(sim)$, $T_i(vis)$ and $T_i(ren)$ directly affect the responsiveness of the environment as they are part of the response time (T_R) equation that was previously discussed. The

amount of each delay depends on the latency of the communication because these (the delays) are often the result of exchanging small-sized parameters between components. For that reason, an IDSV must provide low latent mechanisms for exchanging small data, in addition to using a network with low latency.

The effects of latency might not be obviously apparent for communicating large data volumes. However, behind sending any amount of data through a link, it is often the case where many small packets (i.e. control messages) are primarily exchanged between both ends before the real data transfer is started. So the performance of transferring small messages directly affects the performance of transferring any amount of data. Therefore, latency is a crucial performance metric for a responsive system.

- **Throughput.** One of the main reasons for distribution is to run components on dedicated hardware in order to reduce $T_{sim/vis/ren}$. However, the consequence of distribution is *communication overhead*. Therefore, distribution is useful if the “gain” in $T_{sim/vis/ren}$ is greater than the “loss” in the communication overhead. One way to obtain this is by decreasing the amount of transferred data and by using high bandwidth networks. An IDSV environment must be able to make use of that by fully exploiting the available bandwidth through a high throughput performance, in addition to the use of mechanisms for reducing the amount of transferred data (i.e. data compression).
- **Scalability.** Adding new components (or users) should not result in a performance degradation. For different reasons, there is always a limit at which the performance is going to start decreasing. For instance, the number of components in an IDSV is not known before hand and it is not controlled for flexibility reasons. Moreover, the available resources (hardware, network bandwidth, etc.) is never infinite and therefore, resource depletions are expected to occur for a growing number of clients/components. An IDSV must be able to cope with such a fact and must provide mechanisms to efficiently allocate resources.

2.2.4 Software engineering requirements

- **Simplicity.** The creation of an IDSV should be possible within reasonable amount of effort and time. This issue is always relative as it depends on the underlying problem. The development complexity should not pass certain limits after which IDSV development becomes over engineered.
- **Language and platform agnostic.** Developers must have a large degree of freedom in their choice of programming languages and system architectures. This eases and speeds development, and promotes software reuse.
- **Clear API.** Heterogeneous components need to communicate with each other. This requires them to speak the same language in terms of the “data representation format” and the “interpretation of the data”. A clear API must be provided that allows developers to unify the different data formats into one format that is commonly interpreted by the different components.
- **Reuse.** The environment must support re-usability of components in other IDSV applications. This requirement is a direct result of the two previously discussed requirements.
- **Debugging.** Syntax errors can occur during the development phase and semantic bugs are easy to happen during run-time. These are sometimes hard to discover. For that reason, the environment must prevent the developer from committing semantic violations by offering a debugger to help monitor the execution of the environment.
- **Attribute management.** Despite that coordination is required in IDSV systems, a component also owns crucial attributes that define its state. These attributes must be changed only by that

component. A facility is required that manages *attribute ownership* in order to avoid race conditions. *Attribute publish/subscribe* mechanisms are also required to limit certain communications to members of a restricted group of processes.

2.2.5 Operational requirements

- **Autonomous job deployment/bootstrap.** Some components in the IDSV run as a consequence of actions of other components in the system. A user must be able to initiate/run a collection of components with minimal effort.
- **Resource management.** Distributed resources are shared and used by different groups of people and institutions. The environment must ensure an efficient resource allocation strategy for the components. This depends on the availability, the load and the location of the resource with respect to the location of the component.
- **Single sign-on.** IDSV applications might require the use of resources from different organizations. These resources are shared and governed by different administrative policies that are not willing to compromise their security strategies. For that reason, the IDSV system must support a single sign-on to multiple resources shielding the end-user from the underlying policies of different sites.

2.3 IDSV supporting architectures

Different architectures that have been used to implement IDSV applications are presented in this section. An overview of each will identify their features in the domain of distributed interactive systems and will also identify where they fall short. A selection of an architecture is made at the end based on two criteria: (1) how far the architecture supports the different requirements that were previously discussed and (2) the advantages of using the selected architecture in IDSV environments over other architectures.

2.3.1 Point-to-point coordination using CAVERN

In [4], a socket implementation of an IDSV is described using the *simulated vascular reconstruction* prototype use case. The implementation was based on CAVERN (the CAVE Research Network) [5]. Results showed a great performance mainly in the throughput (average of 100 KB/s) of the system. This is very important because it positively influences the responsiveness of the environment as discussed in section 2.2.3. However, such a rigid implementation is not suitable for an IDSV, because everything is hard-wired. That is, everything is predefined prior to run-time. This gives a great control over the low-level communication but it gives no space for flexibility in the system.

2.3.2 Distributed Interactive Simulation

The Distributed Interactive Simulation (DIS) is a standard that was initially developed for real-time war game simulations over multiple distributed computers [6]. Its application was later extended to the fields of space exploration and medicine. DIS was funded and sponsored by the Defense Advanced Research Project Agency (DARPA) and it was the successor of the SIMNET (SIMulation NETworking) program, which was limited to homogeneous simulators manufactured by the same vendor. DIS extends SIMNET at different levels and supports heterogeneous simulators. Moreover, it was the base on which the High Level Architecture (described in section 2.3.3.2) was born. Some people claim that DIS is dead. However, USAF's Distributed Mission Operations Center (DMOC) continues to advance the DIS protocol. In addition, SISO; a sponsor of IEEE is currently spreading improvement in DIS.

Messages in DIS are grouped into “Protocol Data Units” (PDUs). Each PDU represents a different type of information. For example, the “Entity State” PDU represents information about the appearance, location, velocity, orientation and acceleration of articulated parts of simulated entities. PDUs are transmitted as packets using the UDP protocol. Despite that UDP is not reliable, it was adopted instead of TCP in order to reduce *latency overhead* which is problematic in real-time applications. UDP forces DIS applications to perform packet recovery tasks which are usually the job of the network itself. This is one disadvantage of DIS because it increases the complexity of its applications from different perspectives: design, implementation and maintenance[7]. Another issue in early DIS versions is the use of broadcast to send packets from one participant to all other participants. This has a major overhead because not all messages need to be processed by each participant. As a consequence, message filtering was required at the receiver side. The alternative was to use “static multicast addresses” to group different kinds of PDUs into different communication channels. Static multicast is not adequate for large-scale simulations [8].

Grouping different types of messages into PDUs is inflexible because definition of messages is required *a priori*, in addition to the definition of the type of messages that each participant can process. PDU messages and the use of the UDP protocol makes DIS difficult to develop with.

2.3.3 Linda coordination and communication model

In the mid 80s, David Gelernter from Yale university came up with a new model for parallel programming that he called *Linda* [9]. A model that later was more adopted in the area of distributed systems. According to Gelernter and Carriero, Linda is best defined as a coordination language that forms one of two components that together make up a complete programming language [10]. The concept of Linda is very simple but powerful as well. It provides a “generic” means of communication and coordination between multiple processes that are geographically distributed. Processes in Linda do not explicitly communicate. They send data (i.e. computation requests, computation results, messages, etc.) in the form of tuples to a shared space that is best described as a “blackboard”. These tuples exist in the space independently of any process until they are explicitly withdrawn. Linda provides four basic operations for processes to communicate with the tuple-space:

1. **out**(*t*) to put a tuple in the tuple-space
2. **in**(*template*) to retrieve a tuple from the space. A match between the requested tuple and the template is checked in order to find the tuple.
3. **rd**(*template*) is similar to *in(template)* with the difference that the requested tuple is only read and not removed from the space.
4. **eval**() allows to fork off processes that will do the work in parallel.

This type of communication allows the distribution of processes both in “space” and in “time”. In space, because processes can be geographically distributed. In time, because two processes do not need to be up and running at the same time in order to communicate. A process writes a tuple on the blackboard and leaves. Other processes can make use of that tuple at any time. Therefore, components join and leave at run-time without any prior definitions at the development phase. Developers do not need to define explicit communications between couples of components. Any component can join the tuple-space at any time and make use of existing data.

Developing applications on top of Linda is relatively easy. Developers need to focus on the communication and the coordination between the components and the tuple-space. Explicit communications between processes are not possible. Therefore, changes to one component do not require major changes

to other components. At most, the agreement about the format and the sequence of exchanged tuples must be modified.

Time management, attribute ownership and groups of communicating processes are not offered in a standard way in Linda. However, these can be implemented using the matching mechanisms between tuples and template-tuples of Linda. The complexity of implementing these facilities varies and depends on the complexity of the underlying application.

Persistency is ensured through the concept of Linda itself and it is a decision to be made at the design phase of the application. Unless the user decides to remove the data explicitly, there is no time-limit at which data is removed. This has a major disadvantage that memory might get depleted. Linda in general lacks *garbage collection* mechanisms to remove outdated data. Some of Linda implementations provide mechanisms to remove outdated data from the tuple-space like pyLinda [11].

The following architectures are based on the concept of coordination languages and Linda:

2.3.3.1 SPLICE

SPLICE is a communication architecture developed at Hollandse Signaalapparaten B.V. for distributed embedded systems [12, 13, 14]. It is a data-oriented coordination model in which processes communicate through “agents”. This makes the communication between processes not explicit but mediated by the agents and based on a publish/subscription paradigm. Hence the acronym SPLICE: *Subscription Paradigm for Logical Interconnection of Concurrent Engines*. In SPLICE, a process publishes data that is stored in a “local store” by the process’s agent. The agent then forwards the data to all other agents that manage the processes that have subscribed to that data. The data space that is created is similar to the “tuple-space” paradigm used in Linda.

Three types of data are supported in SPLICE to handle different communication needs between processes. *Periodic* data sorts are data generated repetitively and delivered to active processes at the time the data is published. *Context* data sorts represent the state of the system. *Persistent* data is similar to context data with the difference that it is kept in a persistent store that is a database. The values are retrieved from the database each time a process requires the data.

Time management and attribute ownership are not explicitly implemented in SPLICE but the existing services can be used to implement these.

In [1], an IDSV application was successfully implemented on top of SPLICE. Performance measurements for version 3.8 in that work showed a low transfer rate for messages bigger than 1 MB. The throughput rate dropped from 500 KB/s to around 200 KB/s for a 10 MB message sent between two machines connected by 100 Mb/s network connection. This is due to the limited amount of shared memory available as it is used by SPLICE to store data for communication between processes on the same machine and for communication held by agents between distributed machines. This is a major concern for a responsive IDSV environment where high throughput data transfer is required.

2.3.3.2 The High Level Architecture

The High Level Architecture (HLA) is a general purpose architecture for distributed computer simulation systems. HLA was developed under the leadership of the Defense Modeling and Simulation Office (DMSO) to support reuse and interoperability across the large numbers of different types of simulations developed and maintained by the United States Department of Defense (DoD). It has been accepted as an IEEE standard (IEEE1516) in the year 2000 [15, 16].

An HLA system consists of at least one *federation* that is composed of *federates*. Federations and federates are constructed following a set of defined *rules*. Federates communicate in the form of objects and interactions, and the communication is controlled by a “Run-Time Infrastructure” (RTI). All exchanges of data among federates occur via the RTI. Interaction between federates and the RTI is

governed by an *interface specification*. All objects and interactions between objects in a federation must be described according to an “Object Model Template” (OMT).

HLA is described in [1] for the implementation of an IDSV system. The architecture proved to be suitable for building such an environment. It provides services to solve issues described in section 2.2. It allows communication between heterogeneous systems, it offers various methods to do time management and it supports flexible attribute publish/subscribe and ownership. The performance of HLA for transferring large data volumes proved to be acceptable as well.

HLA has some major disadvantages in the context of an interactive environment. It is “over engineered” and very complex to develop with. Ten rules have to be followed to develop an HLA application. Five of these rules are for “federations” and the other five are for “federates”. For example, federations must have an HLA federation object model (FOM) document in accordance with the OMT. Federates on the other hand, must have an HLA Simulation Object Model (SOM) document in accordance with the OMT. Strictly following the ten rules, HLA becomes inflexible. For instance, federates update any attribute and send/receive interactions as specified in their SOM and they accept and/or transfer ownership of attributes as specified in their SOM as well. That means, all interactions between components must be known and well defined prior to run-time. Moreover, the composition of a federation must be known prior to run-time as well and cannot change during run-time. As a consequence, the system is restricted to a limited set of components. These rules impose a lot of restrictions on the interactivity style required in a dynamic IDSV environment.

2.3.3.3 LinuxTuples

LinuxTuples is a tuple-space implementation written in C and offers Python, C and C++ interfaces to develop clients. It is a direct implementation of *Linda*. That is, it implements its fundamental operations and semantics closely to their origins like *out()*, *in()*, *rd()*, etc. It follows the same matching mechanisms and blocking rules, and provides new operations mainly for non-blocking calls and for monitoring the shared tuple-space. LinuxTuples uses a centralized tuple-space server that runs on Linux machines. It is an open source project initially developed by Will Ware from MIT. LinuxTuples is being used, updated and debugged by Bram Stolk at SARA Computing and Networking Services in Amsterdam [17] in cooperation with the *Scientific Visualization and Virtual Reality* (SVVR) group at the University of Amsterdam [18].

LinuxTuples operations use the TCP/IP protocol for communications between clients and the tuple-space. A TCP connection is opened at each interaction with the tuple-space from the same process. This is a major concern in LinuxTuples as it introduces the overhead of opening and closing a connection. Experiments showed that this overhead does not negatively influence LinuxTuples performance when used on Local Area Networks.

Beside using TCP, LinuxTuples was extended with new operations that use UDP instead. UDP reduces the overhead and provides better performance, but it is unreliable and in the case of LinuxTuples, UDP restricts the size of the transferred data to small datagrams. The unreliable versions of LinuxTuples operations can be used on trusted networks and in cases where the loss of data packets would not drastically affect the underlying application.

LinuxTuples is a very close copy of *Linda*. What was discussed in section 2.3.3 in terms of flexibility, persistency, attribute management and time/space distribution applies to LinuxTuples as well.

2.3.3.4 Related work on Linda

Because of its powerful features and uniqueness among other technologies in distributed computing, people from both the academic and the industrial domains are working on *Linda* in terms of new implementations, different usage contexts and performance enhancements. It is important to introduce some of these implementations and discuss about some major research around this coordination model.

JavaSpaces is a Java implementation of Linda from Sun Microsystems [19, 20, 21]. It is a core Jini service, where processes exchange objects through a space. The most important feature in JavaSpaces is the possibility of exchanging executable content through objects. When an object is read or taken from a space it becomes local to the client who can modify its fields and invoke its methods. A similar implementation to JavaSpaces is TSpaces that is developed at IBM using Java [22]. It provides group communication services, database-like services, URL-based file transfer services and event notification services.

pyLinda is an open-source Python implementation of Linda [11]. It provides some extensions to the fundamental operations (i.e. non blocking), multiple distributed tuple-spaces and a garbage collection mechanism.

Other research and development in the area of tuple-spaces is grouped in three categories. In the first, the focus is on improving the performance of the original Linda implementation by looking at better distribution techniques of the tuple-space and the use of Linda in different contexts. *Lime* uses Linda's model of coordination in mobile environments where tuple-spaces are carried by mobile agents that all together form a federate tuple-space [23]. Another platform that is used in groupware and mobile environments is *L²imbo* that uses mobile agents to bridge tuple-spaces and considers extensive use of IP multicast to provide a fully distributed implementation [24]. In [25] the aim is to form a SuperSpace that could be the basis for a global grid middleware. The implementation is based on JavaSpaces and it uses Transaction Workers for combining independent tuple-spaces.

The second category uses Linda for document-centric communications based on XML documents using standard Web protocols like Hypertext Transfer Protocol (HTTP) and the Simple Object Access Protocol (SOAP). Two important projects *Ruple* [26] and *XMLSpaces* [27] follow this approach. In *Ruple* XML documents are exchanged and documents are queried using an XML query language. *Ruple* did not gain success in the market and therefore, it was canceled. Another implementation in this category is *XMLSpaces* that provides the ability to form tuples from ordinary data-types and XML documents fields together. It uses multiple matching relations for finding tuples in distributed XML spaces.

In the third category, the research is based on the use of tuple-spaces and Web services together. This is motivated by the loose coupling in space and time that Linda provides for distributed applications over the Internet, where previous technologies like CORBA, COM and other commodity technologies have failed. *eLindaWS* provides a tuple-space as a Web service with the primitive operations of Linda published as remotely available operations of that service [28]. The platform was successfully tested using a bioinformatics application. Another important project in this category is *WSSecSpaces* which is similar to *eLindaWS* but with more security using sophisticated matching mechanisms to control access to data and authentication of data producers to protect the shared space [29].

2.3.4 Web services

Web services are a distributed computing technology based on Web standards and open protocols. They allow application-to-application communication over the network and program entities offer their functionality to remote client applications. Web services are becoming increasingly popular in the field of distributed systems and Web applications. For instance, the auction site eBay [30] offers many Web services that can be used by third party developers to embed these into their applications [31]. Amazon Web services [32], Google APIs [33] and many others offer their platforms to the public as Web services.

Three main reasons are behind this success: (1) Web services offer a high level application integration using Web standards and open protocols, (2) they are platform and language independent and (3) they are adequate for Internet-scale applications. In addition to that, Web services allow the implementation of loosely coupled distributed systems; entities at both ends of the wire do not need to be bound to a particular proprietary technology. This promotes software reuse and increases the scalability of the

system as new components can be added without major changes.

Web services capabilities are described in an interface coded with XML using the *Web Services Description Language* (WSDL) [34]. This interface is the identity of the service to the outer world. It tells everything about the service: what functionality it provides, how it works and how to invoke its operations. Messages exchanged between clients and services are wrapped in *Simple Object Access Protocol* envelopes (SOAP) [35]. These are transported over the wire using the *Hypertext Transport Protocol* (HTTP). SOAP is XML-based which means possible to parse on any platform without worrying about interoperability issues.

The major drawback in Web services is their low performance in highly demanding and real-time interactive environments. This is mainly due to exchanging XML data instead of binary code.

2.3.4.1 Web services and IDSVs

Despite that Web services are currently inadequate for IDSV environments, this does not necessarily mean that the situation is not going to change in the future. In fact, key players in information technology have been working together in order to define new standards and protocols to enhance Web services. For example, Microsoft and IBM are the co-founders of the Web Services Interoperability organization (WS-I) for creating specifications. The WS-Addressing specification was submitted to the World Wide Web Consortium by BEA, IBM, Microsoft, SAP, and Sun Microsystems [36, 37]. This in addition to many other joint efforts that are taking place to push the envelope and move Web services forward. It would not be surprising then if real-time components are developed as Web services in the future.

Web services are not restricted to provide direct computational entities (i.e. simulation or visualization components). They are also used as interfaces or portals to computational resources. As a matter of fact, they have appealed the grid computing community [38] where grid services are based on Web services. As an example of that usage, components might need to be authenticated by an *authentication* Web service before acquiring access permission to a certain dedicated hardware. In this context, the low performance of Web services does not matter. What matters is the semantic of Web services in distributed computing.

2.4 Discussion

The features of each communication architecture in IDSV environments are briefly sketched out in an evaluation table. The architecture that is closest to the IDSV requirements is selected and studied further in the next chapter. Table 2.1 summarizes how each of the systems discussed in section 2.3 meets the different IDSV requirements discussed in section 2.2. An evaluation of three pluses on a certain requirement means that the system fully meets that requirement. On the other hand, an evaluation of three minuses means the system does not meet that requirement at all.

Table 2.1: A Summary of how far each communication architecture meets the requirements of an IDSV

	End-User	Functional	Performance	Soft-Eng	Operational
Point-to-Point	++	-	+++	+	-
DIS	++	+++	++	- - -	++
HLA	++	+++	++	- - -	++
SPLICE	++	++	+	- -	+
LinuxTuples	+++	++	++	++	-
Web Services	+	+	- -	++	+

- The Point-to-Point system offers high performance but falls short in terms of characteristics related to autonomous mechanisms, which are necessary in IDSVs to facilitate the useage of the environment.
- DIS and HLA meet most of the requirements imposed by interactive systems. However, they are over engineered and inflexible.
- SPLICE requires developers to implement some facilities like time management and attribute ownership, and does not perform well for transferring large data volumes.
- LinuxTuples is flexible and easy to develop with. It requires developers to implement some facilities as in SPLICE. LinuxTuples performance is acceptable on Local Area Networks, but it imposes some concerns on Wide Area Networks. However, table 2.1 shows a balance in LinuxTuples; what is lost in operational, functional and software engineering requirements is gained in flexibility, ease of use and full (space/time) distribution.
- Web services are not adequate for IDSVs. However, they are ubiquitous and simple to embed their functionality in any software. They are also language and platform agnostic, they offer loose coupling distribution and are suitable for Internet scale applications. They are witnessing major interest in the Web and distributed computing communities, in addition to many joint efforts that are taking place to further enhance them.

Table 2.1 suggests the selection of LinuxTuples as the communication architecture to experiment with in this thesis. Linda systems in general and LinuxTuples in particular have the potential to implement IDSV applications and to support different essential requirements imposed by this type of applications.

IDSV components need access to dedicated resources and these belong to different organizations. Web services are used in many cases as access points to such facilities. From that perspective, it is important for IDSVs to support them. Moreover, despite that Web services are currently of a low performance in real-time applications, the interest they are witnessing from major key players in information technology promises future increases in performance issues.

LinuxTuples is further studied, explained and tested. A bridge model to integrate Web services and Linda-based architectures (using LinuxTuples as an implementation) is investigated. IDSV prototype use cases are implemented.

Chapter 3

LinuxTuples: Characteristics and Evaluation

3.1 Introduction

This chapter introduces and analyses LinuxTuples as the underlying communication and coordination architecture for IDSV applications. The interface (elements and operations) of this architecture is described and several examples are implemented. These examples include different scenarios that help introduce LinuxTuples and identify its features in IDSV environments.

Experiments are performed to evaluate the performance of LinuxTuples in terms of roundtrip time and throughput. These two metrics are important factors to determine the performance of LinuxTuples in different configurations. Configurations where different numbers of components, different networks and different hardware resources are used.

Concluding remarks are drawn at the end of the chapter, in addition to suggestions for extending LinuxTuples with mechanisms that fit the requirements imposed by IDSV applications.

3.2 LinuxTuples

LinuxTuples is a direct implementation of Linda. It implements its basic interface and extends it with additional services. The interface and the elements of LinuxTuples are described and analysed in this section. Common syntax is used when the context applies to Linda systems in general.

3.2.1 Three basic elements

- **Tuple-Space.** This is an abstract computation environment that forms the basis of Linda's model of communication [9]. A tuple-space could be considered as a *distributed shared memory* sometimes referred to as a blackboard that contains tuples. Processes communicate and coordinate through the tuple-space by inserting and/or retrieving tuples.
- **Tuples.** A Tuple is an ordered collection of values which can either be *actuals* or *formals*. For example, in tuple $(1, 'Riad')$ both parameters are actuals, whereas in tuple $(1, firstName:string)$, *firstName* is a formal parameter (can be assigned an actual) and it is used in a template tuple.
- **Template.** A template is a tuple that has a mix of *actual* and *formal* parameters. Example of a template is: $(1, firstName:string)$ which has one actual parameter (i.e. 1) and one formal

parameter of type *string* (i.e. *firstName*). This template matches with tuple $(1, 'Riad')$. Therefore, value 'Riad' is assigned to the formal parameter *firstName*.

3.2.2 Operations

Linda offers a very simple interface that was introduced in section 2.3.3. This interface is implemented in LinuxTuples with the following operations:

- **put**(*tuple*). Puts *tuple* in the tuple-space.
- **get**(*template*). Gets a tuple that matches the *template* input of the operation. In case it is found, the matching tuple is removed from the tuple-space. If not, the operation blocks until one similar tuple is available in the tuple-space.
- **read**(*template*). Same as **get**(*template*) with the difference that the matching tuple is read by the calling process and not removed from the tuple-space.

In LinuxTuples (and Linda in general) it is non-deterministic which process is first served. If two processes request the same tuple at the same time, it is not defined which process gets the tuple first. If both processes issue a *read()* request, both are served one after the other. On the other hand, if the first served process removes the tuple from the space (i.e. using *get()*), the second process blocks until a similar tuple is available.

LinuxTuples does not implement operation *eval()* of Linda. It provides a *jobcontrol script* that offers various facilities such as to run N instances of a program on a cluster, to query what jobs are currently running on the cluster and to stop all the programs distributed on the cluster. In addition to other services to manage and monitor the tuple-space.

To add more features and capabilities to the architecture, LinuxTuples extends the basic interface of Linda with the following additional operations:

- **get_nb**(*template*). A non-blocking get. The process does not block if the requested tuple is not found. The operation returns a NULL pointer.
- **read_nb**(*template*). A non-blocking read. A NULL pointer is returned in case a matching tuple is not found.
- **dump**(). Shows the tuples that are currently in the tuple-space. It also gets a list of tuples that match at least one of a set of templates.
- **log**(). Shows the actions that are taking place in the tuple-space.

All LinuxTuples operations open a TCP/IP connection for each communication with the tuple-space. This adds additional overhead and exhausts the socket address space. Using a large amount of TCP/IP connections causes sockets to go into the TIME_WAIT state. If the number of sockets in the TIME_WAIT becomes too large, the address space is exhausted and TCP packets are dropped. The work in [39] explains the TIME_WAIT problem and provides an alternative that solves this issue for the case of LinuxTuples. However, the solution needs administrative privileges on the host where the TIME_WAIT must be modified.

A *best-effort* version of operation *put()* was added to LinuxTuples by Bram Stolk at SARA Computing and Networking Services in Amsterdam [17]. It uses UDP as the communication protocol that reduces the overhead of TCP, but it is unreliable:

- **put_be**(*tuple*). The *best effort* version of operation **put**(*tuple*). It uses UDP instead of TCP and does a best effort to send the tuple to the tuple-space.

A client/component that needs to communicate with the tuple-space must know the *hostname* on which the space is running and the *port* on which it is listening. These properties can be either explicitly coded in the process or saved in environment variables and automatically read upon opening a connection. The tuple-space is centralized and maintained on a single host.

3.2.3 Matching mechanism

In order to find a certain tuple, the client must make a template that matches the requested tuple. A matching mechanism between tuples and templates is used in Linda systems in order to locate and retrieve tuples.

Let (tpg) and (tpr) refer to the templates generated respectively by operations `get()` and `read()`:

$$\begin{aligned}(tpg) &:= \text{get}('S', 2, j:\text{boolean}) \\ (tpr) &:= \text{read}('group1', f:\text{float}, \text{False})\end{aligned}$$

A possible matching for template (tpg) would be either a tuple of the form $(S, 2, \text{False})$ or $(S, 2, \text{True})$, but nothing else. For example, tuple $(S, 2, \text{'fruit'})$ is not a match since 'fruit' is not of type boolean. A tuple like $(\text{'group2'}, 2.3, \text{False})$ is intended to 'group2' and therefore it does not match with template (tpr).

All *actual* parameters in a template must exactly match their corresponding components in a requested tuple in order for a match to occur. Matching mechanisms are similar in principle to the “select” operation in relational database systems and may be said to make tuples content-addressable [9]. Tuples are not located by memory or identifiers, they are located by their content.

3.3 Examples

A basic math library is implemented using LinuxTuples in order to present the characteristics of the architecture. Additional scenarios are added throughout this section to test the features of LinuxTuples and identify where it falls short.

3.3.1 Add/Subtract functions

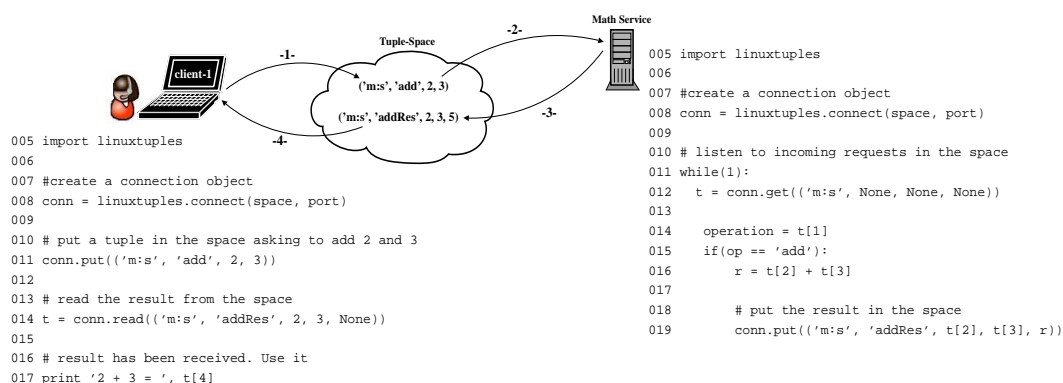


Figure 3.1: A simple illustration of one client that asks for computation services. Math Service responds to these through the tuple-space.

Two operations are provided by the library:

- **add** that adds two integers and returns the result as integer.
- **subt** that subtracts two integers and returns an integer result.

One client is using the tuple-space

Figure 3.1 shows a client and a Math Service that communicate through a tuple-space. The Math Service listens to the tuple-space for incoming requests. This is illustrated through template $(m:s, None, None, None)$ at line 012 of the service code. The template has one *actual* field $m:s$ that designates the name of the service. The other three fields are all *formals*. We go through the scenario represented in the Figure to identify each step in more detail:

1. The client puts tuple $(m:s, add, 2, 3)$ in the space to request the addition of 2 and 3 (see line 011 of the client's code). All fields of this tuple are *actuals*: $m:s$ designates the name of the service, add is the name of the operation and $(2, 3)$ are the numbers to be added.
2. Math Service removes tuple $(m:s, add, 2, 3)$ from the space because it matches with template $(m:s, None, None, None)$. The *actuals* of the client's tuple are assigned to the *formals* of the template in the following order $(m:s, None: add, None: 2, None: 3)$.
3. Math Service computes the addition (see line 016) and puts the result back into the tuple-space as illustrated in line 019.
4. The result is represented in tuple $(m:s, addRes, 2, 3, 5)$. This tuple matches with the template $(m:s, addRes, 2, 3, None)$ that the client reads at line 014.

When the client puts a tuple like $(m:s, add, 2, 3)$ it actually publishes a request. The service subscribes to that request through template $(m:s, None, None, None)$. This is how the *publish/subscribe* mechanism can be implemented in LinuxTuples.

Two more clients joins the tuple-space

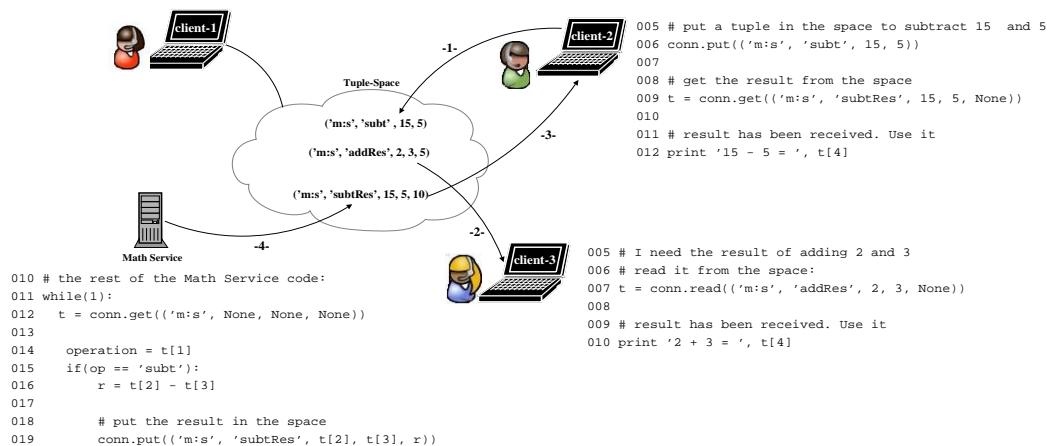


Figure 3.2: Two more clients joins the tuple-space that synchronizes the activities of multiple clients.

1. At line 006, client-2 puts tuple (*'m:s', 'subt', 15, 5*) to request the subtraction of numbers 15 and 5.
2. Because client-1 used *read()* to obtain the result tuple (*'m:s', 'addRes', 2, 3, 5*), the tuple is still in the space and client-3 is able to read it as well (see line 007).
3. Math Service computes the subtraction of 15 and 5 (see line 016) and puts the result into the tuple-space as illustrated at line 019.

Math Service does not need to be up and running at the same time when a client puts a request into the tuple-space. The request is kept in the space until “some” service can handle it.

Client-2 and client-3 joined the tuple-space at run-time without interrupting the course of the application. It was sufficient for the new joining clients to know the format of the exchanged tuples in order to use the environment. No definitions of these clients and of their activities were required *a priori*.

Tuple (*'m:s', addRes', 2, 3, 5*) was used by both client-1 and client-3 that are not aware of each other. That is because of two reasons: (1) the service did not send the response to a particular process, but sent it to the tuple-space. This makes the response equally accessible to all processes. (2) client-1 has read the tuple and did not remove it from the tuple-space. This is how *persistency* (discussed in section 2.2.2) is achieved in LinuxTuples.

LinuxTuples persistency allows *time management* to be implemented. For example, by keeping iteratively generated data into the tuple-space, a process can use results back from any time-step without adjusting the iterative process. However, accumulating data into the tuple-space causes memory depletion in LinuxTuples because the architecture does not offer a *garbage collection* mechanism to clean the tuple-space from outdated tuples.

3.3.2 An average function

The library is extended with a function that computes the average of a group of integer numbers as shown in Figure 3.3. This helps identify how *attribute ownership* can be implemented in LinuxTuples.

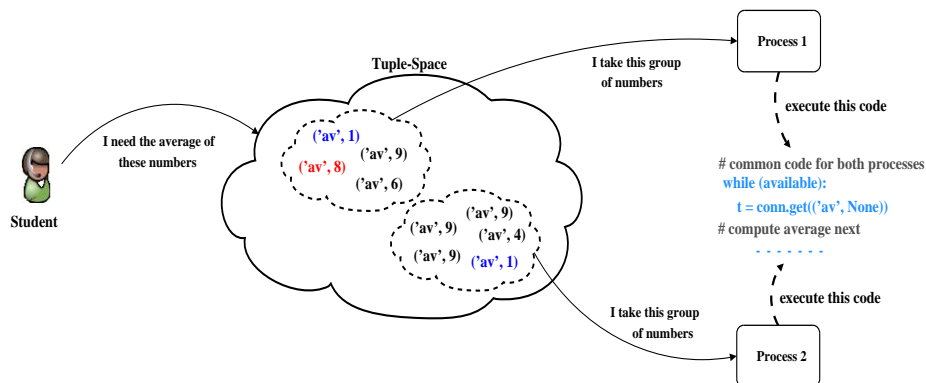


Figure 3.3: “Process 1” and “Process 2” both participate in computing the average of the numbers in the tuple-space. To ensure a correct result, both processes use operation *get()* to remove tuples from the space.

The problem presented in Figure 3.3 is simple; if a number appears once, it must be taken by one and only one process. That is the case for tuple (*'av', 8*). If both processes take a copy of tuple (*'av', 8*), the result of the average becomes wrong. To ensure a correct result, both processes use operation *get()* to remove the tuple from the space and make sure it is taken by only “one process”. The solution is

successful as illustrated in Figure 3.3. Tuple $(av, 8)$ is taken by “process 1” ensuring the coherency of the data. On the other hand, each process got one copy of tuple $(av, 1)$. This is not a problem, because number 1 appears twice between the numbers that “student” has put into the space.

3.3.3 A frequency function

The implementation of *attribute ownership* is not always simple in LinuxTuples as in the previous example. The complexity of implementing *attribute ownership* depends on the complexity of the underlying problem. The math library is extended with a new operation that computes the frequency of numbers. The challenge that LinuxTuples faces in this example is when multiple processes participate in the frequency computation as illustrated in Figure 3.4.

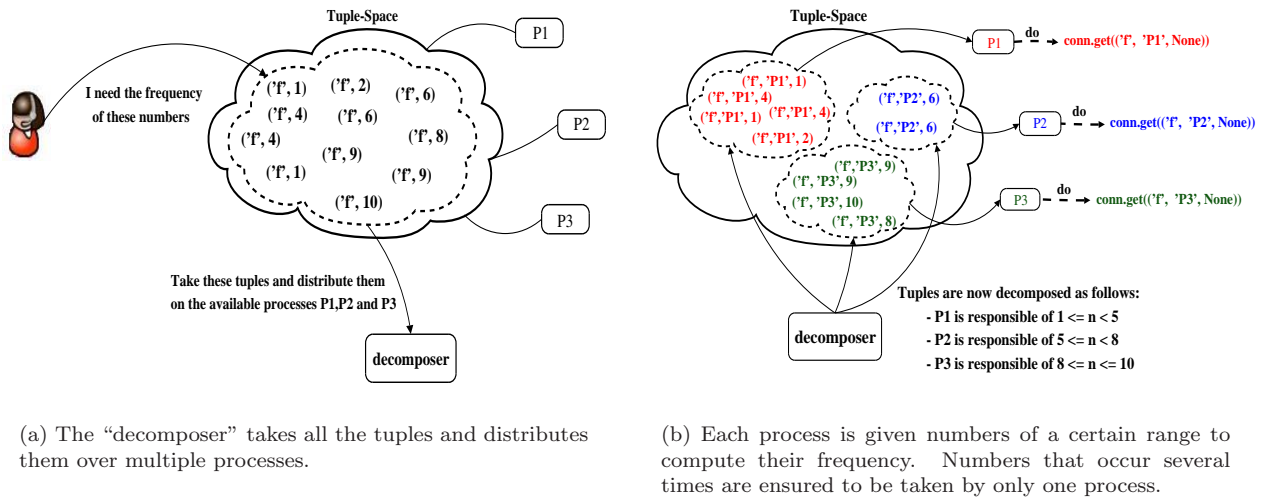


Figure 3.4: *Attribute ownership prevents occurrences of the same number to appear at different processes.*

All occurrences of the same number must appear at one and only one process. Otherwise, each process computes a different frequency of the same number. A correct result is ensured if *attribute ownership* is applied. In Figure 3.4(b), the decomposer distributes numbers of different ranges to each process. For instance, “P1” is responsible to compute the frequency of numbers ranging from 1 to 4, “P2” computes the frequency of numbers between 5 and 7 and “P3” takes numbers between 8 and 10. The decomposer first needs to check how many processes are available at the moment the request is made. Second, it needs to distribute the numbers on all processes. This is done with `conn.put(('f', 'Pi', number))`, where P_i is the name of a given process with its number i . Each process accesses the tuples that it is restricted to access by doing `conn.get(('f', 'Pi', None))`. This ensures the coherency and the correctness of the end-result.

This example reveals a hidden problem. If one of the participant processes fails for any reason (i.e. hardware failure), two situations can occur:

- If the process fails before it removes the tuples from the space, the process can make use of the tuples when it is restarted.
- If the process fails after removing the tuples from the space, the tuples are lost. LinuxTuples does not offer any mechanism to recover lost tuples. The alternative is to use operation `read()` instead of `get()`, but this leads to another potential problem which is memory depletion. LinuxTuples does

not offer any *garbage collection* mechanism to remove outdated data from the space. The only way to remove data in LinuxTuples is through the application. This might work for the basic math library but it is an inflexible method for IDSV environments where the number of components and their activities are non-deterministic.

3.4 Performance analysis

IDSV applications require a high performance architecture that ensures instant communications between components and real-time feedback to end-users' interactions. Performance is also crucial for the usefulness of IDSV environments in terms of providing correct and comprehensive information. The performance of LinuxTuples is studied and analysed in this section through different measurements.

3.4.1 Roundtrip time

Components in IDSVs interact by exchanging parameters. The delay between the moment an interaction was initiated and the moment it is received by the target component influences the responsiveness of the application. Therefore, events or parameters sent back and forth between components must be received in a minimal period of time. Hence the importance of measuring the roundtrip time of LinuxTuples. By roundtrip time we mean the delay between the moment a message is sent to the tuple-space and the moment it is received back. For that purpose, a simple “pingpong” script is used that allows clients to exchange messages through the tuple-space. Figure 3.5 presents a snapshot of the script and a scenario of two clients playing pingpong through the tuple-space.

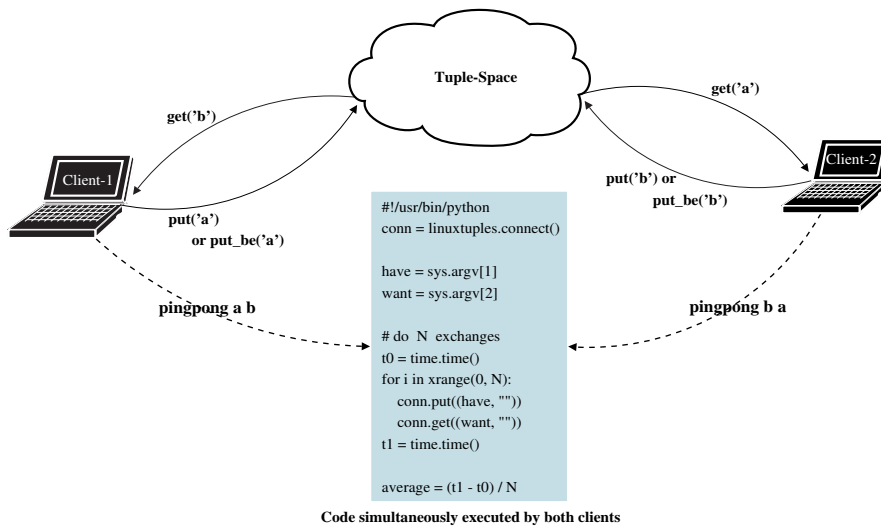


Figure 3.5: A snapshot of the “pingpong” code and a possible scenario of a single pair of clients.

Two versions of “pingpong” are used: (1) a *reliable version*, where clients use *put()/get()* operations to send/receive messages. (2) a *best effort version*, where *put_be()/get()* operations are used for the same purpose.

All experiments were performed using the SARA HP-Graphics cluster. This cluster contains 29 nodes each with two Intel Xeon processors running at 3.4 GHz and 2 GB memory. Each node uses an Intel

Gigabit Network Interface Card (NIC) with large (9000 bytes) Maximum Transmission Units (MTUs). All nodes are connected to a Cisco Catalyst 4948 switch. This switch has a 96 Gb/s nonblocking switch fabric. Also, it has 2 ports of 10 Gb/s for external connectivity. Moreover, all nodes have a Gb/s full duplex connectivity to all other nodes.

3.4.1.1 Roundtrip time using reliable communication

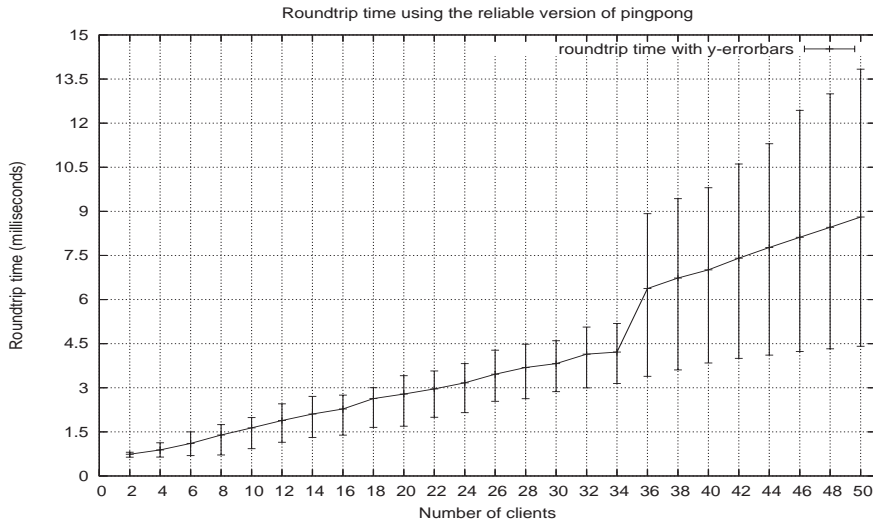


Figure 3.6: Roundtrip time for an increasing number of clients. The y-errorbars represent the maximal/minimal roundtrip times achieved in each experiment.

One tuple-space server handles the clients. We expect the roundtrip time to increase when the number of clients increases. This is illustrated in the measured results of Figure 3.6. The roundtrip time increases linearly at a 0.11 ms rate for each additional client until 34 clients. A bigger jump of around 2.1 ms occurs at 36 clients (~ 1.05 ms per client). Afterward, the graph goes linear again with around 0.15 ms increase per client.

It is clear that resources (i.e. TCP sockets) start getting depleted when more than 34 clients are simultaneously using the tuple-space. This is illustrated by the difference between the minimal and the maximal roundtrip time (y-errorbar) that increases starting from 36 clients. This is due to outlier occurrences in the roundtrip times caused by processes that block while they try to acquire resources.

A general model

A model is provided in order to predict the roundtrip time of LinuxTuples in different configurations. That is, configurations with different networks, different computational resources, different number of clients and different exchanged data volumes.

Definitions:

$t_{C(cl)}$: client computation time.

$t_{C(sp)}$: space computation time.

t_{open_tcp} : time to open a TCP connection.

t_{TCP_RT} : TCP roundtrip latency for a “zero” size message to go back and forth between two nodes.

N_{bw} : network bandwidth.

Network latency: $t_{Lat_N} = t_{open_tcp} + t_{TCP_RT}$

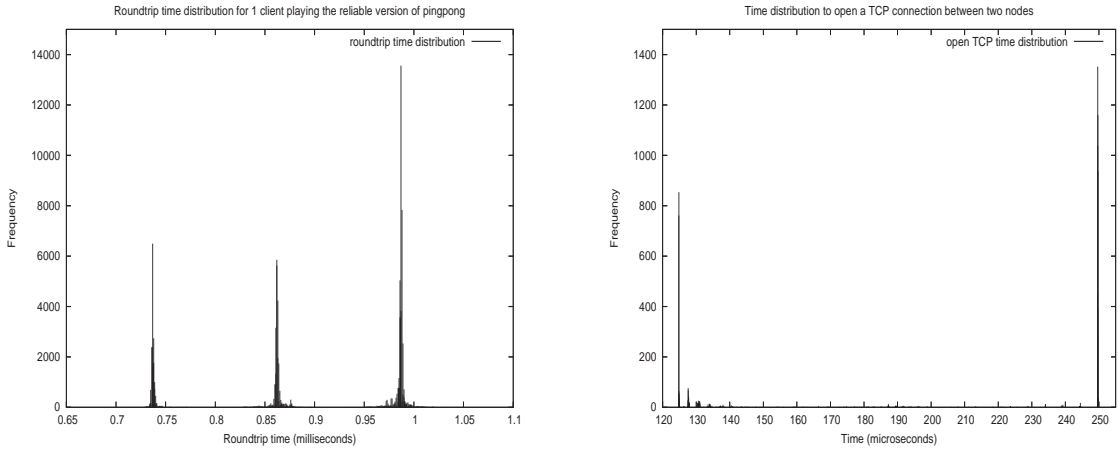
Operation latency: $|t_{Lat_op}| = t_{Lat_op} - t_{Lat_N}$

LinuxTuples roundtrip time $t_{LT_Roundtrip}$ using a reliable (TCP) means of communication can be described by equation 3.1:

$$t_{LT_Roundtrip} = t_{C(cl)} + t_{Lat_N} + 2 * |t_{Lat_op}| + \frac{MessageSize}{N_{bw}} + t_{C(sp)} \quad (3.1)$$

Variable $|t_{Lat_op}|$ is multiplied by two on account that a client uses two operations to put/get a message from the space.

The network latency plays an important role in equation 3.1. This is expected since LinuxTuples extensively uses TCP to communicate with the tuple-space. The influence of opening a TCP connection for each communication becomes more apparent when the distribution of the roundtrip time is plotted along with the distribution of t_{open_tcp} :



(a) Roundtrip time distribution. This corresponds to the measurements of Figure 3.6 for a single client.

(b) Time distribution to open a TCP connection between two nodes on the cluster.

Figure 3.7: Figure (a) shows the roundtrip time distribution for one client playing pingpong through the tuple-space. Figure (b) shows the time distribution to open a TCP connection (t_{open_tcp}) between two nodes on the cluster.

Three different distributions for the roundtrip time appear in Figure 3.7(a). These are explained in Figure 3.7(b) that shows two main distributions to open a TCP connection. One distribution at 125 microseconds and the other at 249 microseconds. If the time it takes to open a TCP connection fluctuates between two distributions, the network latency $t_{Lat_N} = t_{open_tcp} + t_{TCP_RT}$ is expected to fluctuate as well. This in consequence affects the roundtrip time of LinuxTuples. Hence the importance of the network latency in equation 3.1.

In addition to the network latency, the computation time $T_{C(sp)}$ at the tuple-space server influences the

responsiveness of the system as well. That is due to the use of a single centralized tuple-space server to synchronize the communication and coordination between multiple clients that compete for the same single resource.

3.4.1.2 Roundtrip time using best effort communication

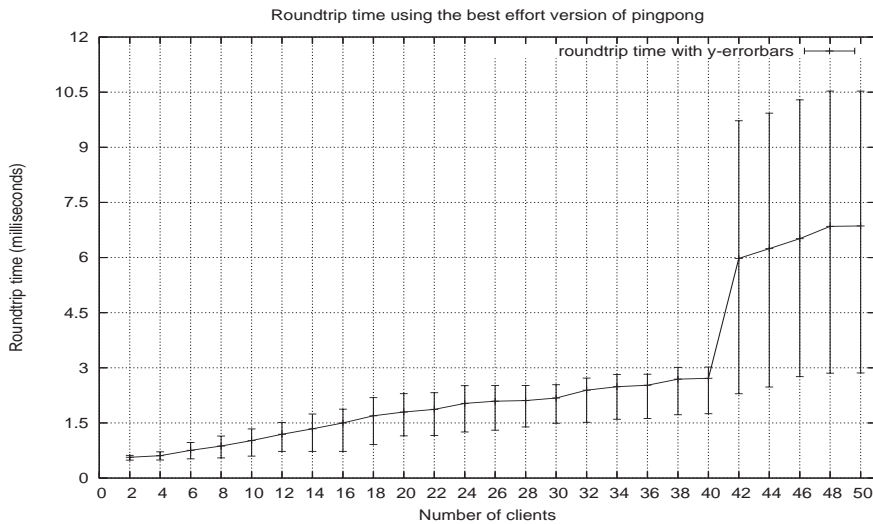


Figure 3.8: Roundtrip time for an increasing number of clients. The y-errorbars represent the maximal/minimal roundtrip times achieved in each experiment.

The roundtrip time increases linearly at a 0.04 ms rate for each added client until 40 clients. The use of the “best effort” version of pingpong has delayed the effect of resource depletion until 42 clients. At that point, a jump of 3.27 ms (~ 1.63 ms per client) occurs. Afterward, the roundtrip time increases linearly again. When resources are depleted at 42 clients, the spread in the roundtrip times increases as illustrated by the difference between the minimal and the maximal roundtrip time (y-errorbar) achieved in each experiment. This is due to outlier occurrences in the data caused by clients that block and wait for resources to become available.

A general model

A model is provided to give an indication of LinuxTuples roundtrip time in different configurations. This model applies when `put_be()/get()` are used to send/retrieve messages from the tuple-space. It is important to recall that operation `put_be()` uses UDP to communicate with the server while operation `get()` uses TCP.

Definitions:

- $t_{C(cl)}$: client computation time.
- $t_{C(sp)}$: space computation time.
- t_{open_tcp} : time to open a TCP connection.
- $t_{Lat_TCP_1W}$: TCP one-way latency for a “zero” size message to go in 1 direction between 2 nodes.
- $t_{Lat_UDP_1W}$: UDP one-way latency for a “zero” size message to go in 1 direction between 2 nodes.

N_{bw} : Network bandwidth.

TCP network latency: $t_{Lat_TCP_N} = t_{open_tcp} + t_{Lat_TCP_1W}$

UDP network latency: $t_{Lat_UDP_N} = t_{Lat_UDP_1W}$

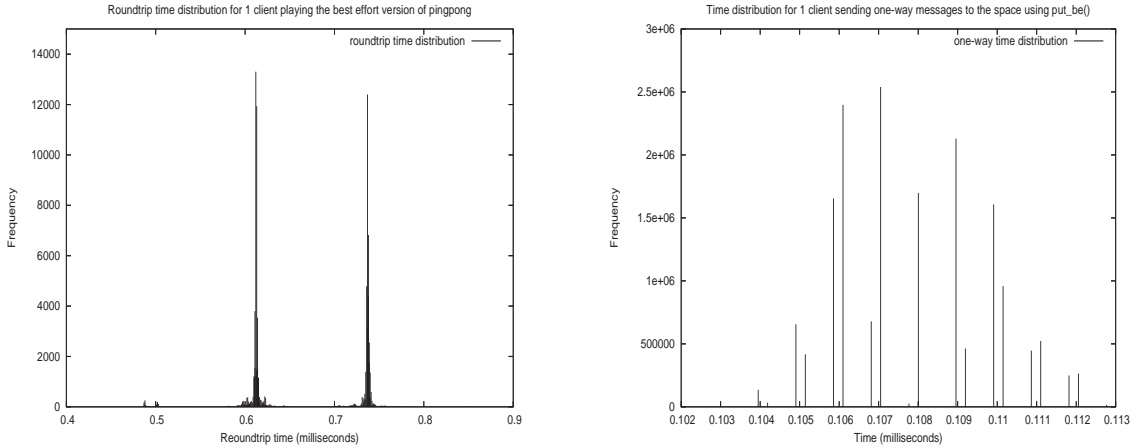
Operation latency (put_be): $|t_{Lat_put_be}| = t_{Lat_put_be} - t_{Lat_UDP_N}$

Operation latency (get): $|t_{Lat_get}| = t_{Lat_get} - t_{Lat_TCP_N}$

LinuxTuples roundtrip time $t_{LT_Roundtrip}$ can be described by equation 3.2, which applies when clients use the UDP protocol to put messages in the space:

$$T_{LT_Roundtrip} = t_{C(cl)} + t_{Lat_TCP_N} + t_{Lat_UDP_N} + |t_{Lat_put_be}| + |t_{Lat_get}| + \frac{MessageSize}{N_{bw}} + t_{C(sp)} \quad (3.2)$$

There is a different latency measurement for each operation in the model above, because each uses a different communication protocol. The TCP latency is higher than the latency of UDP and its presence influences the patterns of the roundtrip time as shown in Figure 3.9(a) below:



(a) Roundtrip time distribution. This corresponds to the measurements of Figure 3.8 for a single client.

(b) One-way-trip time distribution. A single client uses `put_be()` to send messages to the tuple-space.

Figure 3.9: Figure (a) shows the roundtrip time distribution for one clients using the best effort version of pingpong. Figure (b) shows the time distribution for one client sending messages to the tuple-space using `put_be()`.

Despite the use of operation `put_be()`, two distinct distributions are shown in Figure 3.9(a). This is due to the use of operation `get()` to retrieve tuples from the space. `get()` uses TCP and that explains the presence of the two distributions.

If we eliminate TCP from the communication, the time distribution becomes closer to a normal distribution. This is shown in Figure 3.9(b). The graph presents a one-way-trip time distribution when a single client is only using `put_be()` to send messages to the tuple-space without retrieving them. The data has an average (mean) of 0.1078 ms, and a standard deviation of 0.002.

3.4.2 Throughput

In this section we analyse the throughput of LinuxTuples operations for sending data between clients and the tuple-space. Operations responsible for transferring data in LinuxTuples are *put()*, *put_be()*, *get()* and *read()*. Measurements have shown that *get()* and *read()* share the same throughput performance, so results for *get()* are excluded in this section. Operation *put_be()* does not allow the transfer of large data volumes, because messages are sent at once in order to avoid the overhead of decomposing large messages in small datagrams. For that reason, throughput measurements for this operation are excluded as well.

All measurements were performed between two desktops at the UvA-CSP lab in the science park. These are two identical Intel(R) Pentium(R) 4 CPU 2.80 GHz using 512 KB cache size. Both are running Fedora core 2.6.12-1.1381_FC3smp Linux. Each desktop is equipped with an Intel Corporation 82540EM Gigabit Ethernet Controller connected to a 100 Mb/s network.

The TCP throughput between these two nodes is measured with *lmbench* [40]. For messages smaller than 1 MB, TCP throughput is 95 MB/s. For messages equal to or greater than 1 MB, TCP throughput is 105 MB/s. This helps identify the overhead of each operation.

Figure 3.10 plots the throughput of operations *put()* and *read()*, in addition to the TCP throughput:

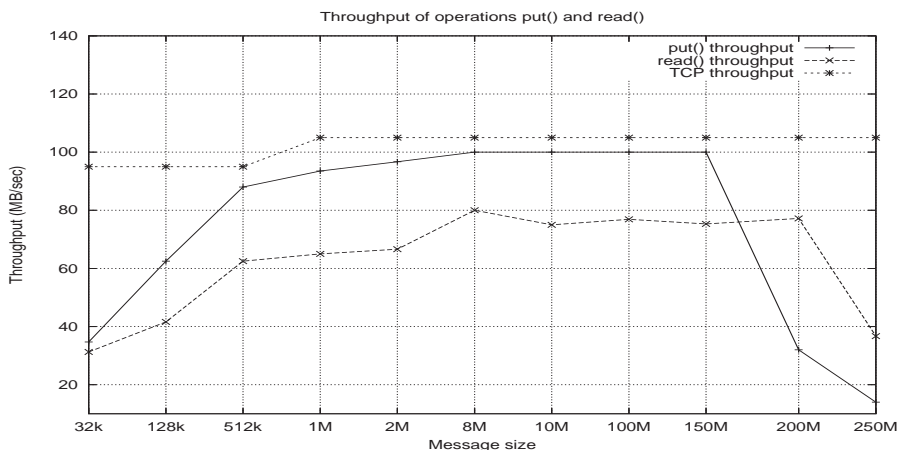


Figure 3.10: *Throughput of operations put() and read() measured for different message sizes.*

In the case of *put()*, the throughput increases when the message size increases. It goes stable at 100 MB/s for messages between 8 MB to 150 MB and drops to 32 MB/s for 200 MB message and to 14 MB/s for a message of 250 MB. Using Figure 3.10 and the TCP throughput, we can draw the following analysis represented in table 3.1:

Table 3.1: *Throughput analysis of operation put().*

Message size: M	TCP Throughput	put() Throughput	Overhead
$M = 1 \text{ MB}$	105 MB/s	93.5 MB/s	11 %
$8 \text{ MB} \leq M \leq 150 \text{ MB}$	105 MB/s	100 MB/s	5 %
$M = 200 \text{ MB}$	105 MB/s	32 MB/s	70 %
$M = 250 \text{ MB}$	105 MB/s	14 MB/s	87 %

When messages are smaller than 1 MB, the headers that are sent along with each message introduces an overhead and causes a low throughput. These headers on the other hand are too small to affect larger messages (i.e. between 1 MB to 150 MB).

As for operation *read()*, the shape of the throughput line is more or less similar to the one for *put()*. However, the upper throughput limit for *read()* is 80 MB/s. Table 3.2 represents the throughput analysis of operation *read()* using Figure 3.10 and the TCP throughput:

Table 3.2: *Throughput analysis of operation read()*.

Message size: M	TCP Throughput	read() Throughput	Overhead
$M = 1$ MB	105 MB/s	65 MB/s	38 %
$M = 8$ MB	105 MB/s	80 MB/s	24 %
$M = 200$ MB	105 MB/s	77 MB/s	27 %
$M = 250$ MB	105 MB/s	37 MB/s	65 %

The overhead only increases from 24% for reading a 8 MB message to 27% for a 200 MB message. Therefore, *read()* has better throughput than *put()* for messages larger than 150 MB/s.

It was expected that *read()* would have a 100 MB/s upper limit throughput similar to *put()*. Though it is not the case. The template that operation *read()* uses is matched by the server in order to find the requested tuple. This may be the cause for this lower throughput upper limit.

3.5 Conclusion

LinuxTuples is a close implementation of Linda. It extends its basic interface with additional operations for more flexibility and to handle additional application scenarios. Moreover, LinuxTuples clients can be implemented in two programming languages: Python and C++. The examples introduced in this chapter identified the following features of LinuxTuples:

- The communication in LinuxTuples is generic. Components communicate through the tuple-space.
- The tuple-space is a persistent store of data. Tuples remain in the space until they are explicitly removed by a process.
- LinuxTuples distributes components in space and time. These can join the tuple-space at run-time without the need to define any communications *a priori*. Though, the components must agree on the format of the exchanged tuples.
- LinuxTuples does not offer direct solutions to do *time management*, *publish/subscribe* and *attribute ownership*. However, these can be implemented using existing LinuxTuples services.

It is relatively easy to implement applications using LinuxTuples. The interface is clear and easy to use. However, the examples introduced in this chapter revealed that LinuxTuples lacks necessary mechanisms to cope with complicated scenarios. This requires developers to perform extra work at the application level. The following are some of these possible scenarios:

- A scenario where component C puts tuples into the space for multiple components to use. These tuples must persist. However, component C might produce new results and therefore old tuples must be removed in order to prevent memory depletion. The only way to remove tuples in LinuxTuples is through operation *get()*. This has two consequences: (1) It must be hard coded

by the developer, which is inflexible because time-events in IDSVs are non-deterministic. (2) A *get()* operation per each tuple in order to clean the space is additional overhead. An alternative must be implemented to automate the process of cleaning the tuple-space from outdated tuples. Adding a timestamp to tuples can be a solution.

- LinuxTuples uses a centralized tuple-space server without any failure recovery mechanism. This is dangerous since any failure of the server that hosts the tuple-space results in the loss of all existing tuples. This has two consequences: (1) persistency is compromised and (2) applications using the tuple-space fail. Using multiple redundant distributed tuple-spaces is an alternative to keep multiple copies of the tuples.
- If a client that offers services to other components through the tuple-space fails for a certain reason (i.e. hardware or network failure), there is no way in LinuxTuples to signal that failure to all components that depend on that client. Moreover, if the client recovers, LinuxTuples does not offer any roll-back mechanism for the client to continue from the point where it has stopped.

Automated mechanisms must be implemented in LinuxTuples to provide solutions for scenarios such as the ones listed above.

Performance measurements for LinuxTuples operations were presented. These include the roundtrip time for an increasing number of clients and the throughput for transferring data from a client to the tuple-space and vice versa. Results showed that resources start getting depleted at 36 clients using a reliable communication (*put/get*). This is related to the use of a single centralized tuple-space and to extensively opening TCP/IP connections. In the case of the best effort communication (*put_be/get*), the resources start getting depleted at 42 clients. This slight increase in performance is due to operation *put_be()* that uses the UDP protocol. A model was provided to predict the roundtrip time of LinuxTuples on different configurations (networks, number of clients and computational resources). The model showed that LinuxTuples responsiveness depends on the underlying network, specifically the latency of the network. The latency increases as the distances between remote hosts increase. This raises a major concern about the performance of LinuxTuples on Wide Area Networks.

Chapter 4

Integrating Web services and Linda

4.1 Introduction

IDSV environments are meant to support computationally intensive applications that require massive computing power, large data storage, fast networks and advanced graphic cards and display devices. Applications of that character belong to the “e-Science” world, where large geographically distributed teams come together to collaboratively solve complex problems by sharing their knowledge and their computational resources [41]. From that perspective, the distributed components that form the underlying application are offered by different groups and therefore, they originate from different technologies. Web services are one of these software system technologies for developing distributed computational services and interoperable machine-to-machine communications [42]. It is widely used and attracts the interest of an increasing number of different application domains [43]. This is due to many reasons, some of which were discussed in section 2.3.4.

Despite that Web services are currently inadequate for IDSVs, the joint efforts and interest toward this technology promise a change in the future. A change that would make developers implement scientific components using Web services. Therefore, an IDSV application built on top of a Linda communication architecture might have one or many of its components developed as Web services. As a consequence, a communication between these components and Linda systems becomes required.

In the field of “e-Science” collaboration and access to high performance computing power are required. Web services are currently used as interfaces or access points to computational resources. For instance, the grid computing infrastructure that brings researchers and scientists together to achieve “collaborative problem solving” offers its services as Web services [44]. This raises another motivation for a communication between Linda and Web services in order for IDSV components to access dedicated resources.

A bridge model that integrates Linda and Web services is studied in this chapter. A prototype is proposed as a possible solution, requirements are discussed and an example is implemented. Results are presented and the chapter ends with discussion and concluding remarks.

4.2 An introduction to Web services

Web services are a software technology that allows the development and integration of distributed applications that share multiple resources (i.e. computational resources, networks, software, etc...) [45]. They use open protocols and general-purpose standards like HTTP, SOAP and XML to communicate between geographically distributed software components. In this section, a simple test case is used to

illustrate the basic mechanisms behind a Web service.

Consider a school that offers a *math library* to students in order to support them as a tool to do their homework. The functions of the library are implemented as remotely accessible operations within a Web service. To keep it simple, the library is reduced in this example to a single operation: *add()*. The operation adds two numbers and returns the result back to the user. Figure 4.1 presents the required steps for a client application to successfully invoke the *Math-Service*.

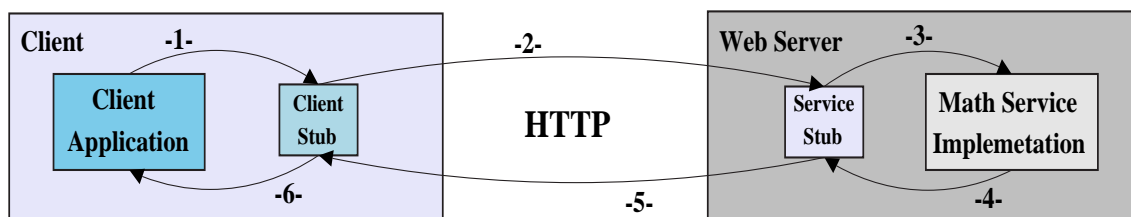


Figure 4.1: *Mechanisms behind invoking a Web service. Picture based on [46].*

To use a Web service, the client must locate it first. The address of a Web service is represented in a Uniform Resource Identifier (URI) that points to the location of the service. In practice, the URI is provided by a discovery service [47]. Suppose that *Math-Service* is successfully located, the client obtains a WSDL interface file that describes the service. The WSDL describes the input/output parameters of the operations and how to invoke them, in addition to other information like communication protocols and specifications used by the service. After understanding the service, the client can invoke its operations. In this example the client invokes operation *add()* (see Figure 4.1) as follows:

1. To invoke *add()*, the client application delegates this task to the *client stub*. The stub makes a SOAP request document out of the client's local invocation. This process is called *marshaling* or *serializing*.
2. The *client stub* sends the SOAP request over the network using HTTP. At the other side, the *service stub* receives the request and deserializes it so the service implementation can deal with it.
3. The *service stub* hands the deserialized SOAP request to the service implementation that starts handling the work it is supposed to do: the addition of two numbers.
4. When the service implementation has finished adding two numbers, it hands the result back to the *service stub*. The *service stub* wraps the result into a SOAP response document.
5. The *service stub* sends the SOAP response to the client. The *client stub* receives the message and deserializes it for use by the client application.
6. The *client stub* hands the final result to the client application.

It is important to note that the client and the service stubs are both generated from the WSDL interface description of the service. Dedicated tools are offered by each Web service toolkit to automatically generate stubs. One such example is the *ant* [48] tool.

The scenario discussed in this section presents the basics behind the use of Web services. However, Web services offer more advanced functionality for more complicated distributed applications.

4.3 Differences between Web services and Linda

The consequence of integrating Web services and Linda-based architectures raises many problems because of the differences between these two architectures. These are semantic and syntactic differences that are summarized through the following points:

- Linda distributes processes in space and time. On the other hand, Web services allow distribution in space only. Therefore, two processes in Web services must be up and running at the same time in order to communicate.
- In Linda, the communication and the computation are two separate processes. Linda only provides an interface through which processes can communicate with tuple-spaces. Computation tasks cannot be implemented with Linda. In Web services on the other hand, the computation and the communication are two coupled processes. Developers must adapt the application to the underlying communication architecture in order to publish its functionality to the outer world.
- Web services are not persistent. That is, an instance of a Web service is created only when a client needs to use the service. On the other hand, persistency is inherited from Linda's concept. Tuple-spaces run independently of any other processes.
- Both technologies represent data using different representation formats. In Linda processes exchange tuples through the tuple-space. A tuple is a collection of arbitrary typed-fields. In Web services data is represented in SOAP which is XML-based and must be parsed to extract the required information.
- Web services use HTTP to transfer data (SOAP messages) over the network. In Linda the transport protocol depends on the underlying implementation. For example, LinuxTuples uses plain TCP and UDP to transfer data over the wire.

4.4 A bridge between Linda and Web services

A bridge model to connect Web services and Linda is studied in this section. We discuss the approach for developing the model and the requirements that a generic bridge must implement. An example prototype is implemented and the work is discussed at the end of the section.

4.4.1 Approach

Since Web services and Linda are semantically and syntactically different, a direct communication between them is impossible. The alternative is a “broker” that understands both technologies and therefore, can mediate the communication between them. Figure 4.2 presents a bridge model that connects a Web service to a tuple-space using a broker:

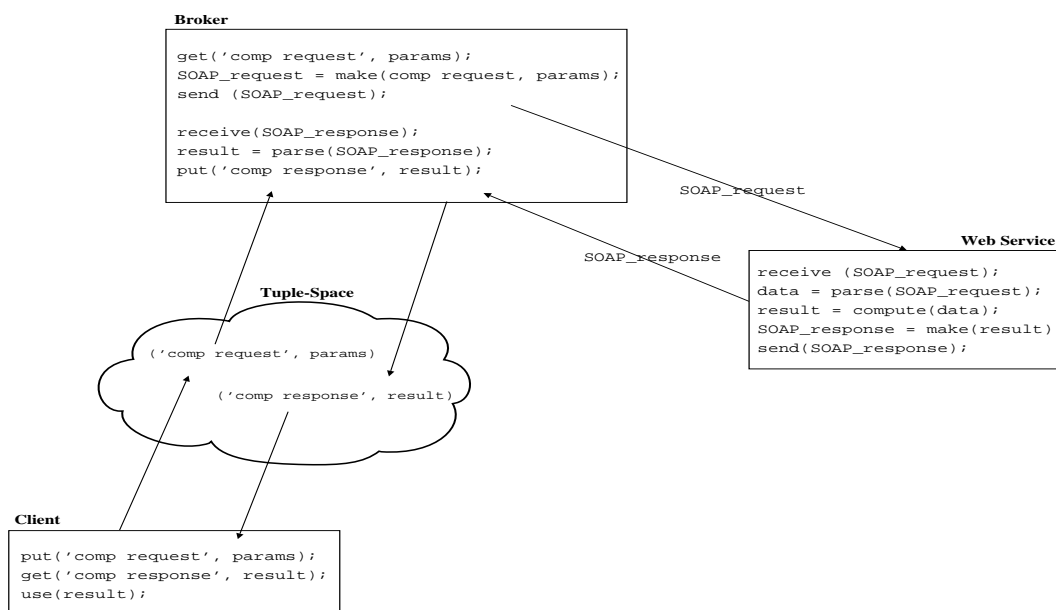


Figure 4.2: A bridge model that connects a Web service to a tuple-space.

The broker must offer the following facilities in order to be capable of integrating a Web service and a Linda architecture:

- A Web service discovery mechanism to discover available Web services. After discovery, the broker must access the service's WSDL file that describes its characteristics.
- The broker must parse the WSDL file and understands how the service can be used. This is necessary in order to send SOAP requests to the service and receive SOAP responses back.
- The broker must translate the content of tuples into SOAP requests for the service to handle. This requires the broker to extract necessary parameters from the tuple and wrap these into a SOAP message document to make the correct Web service invocation.
- The broker must translate a SOAP response into a tuple and put this into the tuple-space. The format of the tuple must adhere to the application logic. That is, beside the result that was sent by the service, *actual* parameters are also required into the tuple in order to satisfy a match between this tuple and the template that the client application is expecting.

The broker can be developed using a Web services toolkit (or WS-toolkit) that implements these facilities. This is the approach used in this chapter to integrate Web services and Linda systems.

Web services are not limited to sending and receiving SOAP messages for simple remote method invocations. Scenarios and the data exchanged between Web services can become more complicated. For that reason, the WS-toolkit must meet different requirements in order to provide full capabilities to the broker and common methods of using Web services. These requirements are discussed next.

4.4.2 Requirements

Web services' support for standards and specifications makes them an interoperable technology for implementing distributed systems comprising heterogeneous software and hardware. This architecture

is not limited to invoking remote operations that exchange primitive data-types. In fact, Web services scenarios can be more complicated. Therefore, mechanisms and standards are required for implementing these complicated scenarios. For example, a notification mechanism is required in order to notify all clients of changes to the Web service. For that reason and in order to provide a robust “broker” that successfully deals with different complicated applications, the WS-toolkit must implement basic mechanisms and standards, and must support common Web services specifications as follows:

1. **Basic SOAP library.** The toolkit must offer basic Web services functionality like creating client and service stubs from WSDL interfaces to handle SOAP requests and responses.
2. **Support for different languages.** To communicate with different Linda implementations, the toolkit must provide interfaces to different programming languages.
3. **Support for complex data types.** A WS-toolkit must support mappings of complex data between native and SOAP data types. For example, a service and a client might exchange multidimensional arrays. The toolkit must provide an API that successfully maps the representation of the array from SOAP to a native representation and vice versa.
4. **Attachments.** Some messages are too large to be embedded in a SOAP envelope and some data types are not XML friendly to reside in a SOAP message. The alternative is to use attachments. With attachments, data (i.e. images) is attached to and referenced from the SOAP message while keeping it in its original format. It is also faster to send large data volumes in attachments rather than sending it in plain SOAP.
5. **WS-Addressing.** A specification accepted by W3C in August 2004 [37]. It provides a transport-neutral mechanism and a more versatile way of addressing Web services than plain URIs. It also helps build Web services on top of a variety of messaging systems [49].
6. **WS-Notification.** To report changes in a Web service. A client can be configured as a *notification consumer* and the Web service as a *notification producer*. The changes that the developer has specified are notified to all client subscribers.

Grid services have converged toward Web services under the *Web Services Resource Framework (WSRF)* specification. This makes grid resources accessible within a Web services architecture. If IDSV components need access to grid resources, the broker must support the WSRF family of specifications. This includes four specifications:

1. **WS-ResourceProperties.** WSRF uses the notion of *resources* to keep state. Every resource is composed of zero or more *resource properties*. For example, a resource *human* has *weight*, *height*, *hair color*, etc as resource properties.
2. **WS-ResourceLifetime.** To manage the life cycle of resources.
3. **WS-ServiceGroup.** To manage groups of Web services. It provides mechanisms to find services, to insert a service into a group and to remove a service from a group.
4. **WS-BaseFaults.** To report faults when things go wrong in a certain Web service invocation.

A Python WS-toolkit is introduced in the next section and used to implement a broker that mediates the communication between a tuple-space and a Web service in a simple test case.

4.4.3 The SOAPpy toolkit

SOAPpy is an easy to use SOAP library for Python and it is one of the toolkits integrated into the Python Web Services Project [50]. SOAPpy supports WSDL and interactions between Python clients and Web services. Through SOAPpy, clients are able to invoke “remote” Web services operations as if they are doing “local” method invocations. This abstraction shields developers from all the burden of XML settings and from the underlying communications. With SOAPpy clients can either directly use a *proxy class* that takes the URI of the service as input or a WSDL *proxy class* that takes the WSDL file as input. In the latter case, a *proxy class* is internally created that uses the URI of the service specified in the WSDL file. The *proxy class* handles all the mechanisms for invoking the service; it creates the SOAP request document, it sends the request over HTTP to the Web service, it parses the SOAP response and creates native Python values for the client to use.

The client can as well inspect what operations the service provides, the type and number of input parameters they require and their output results. These are exposed by the WSDL *proxy class* as a Python dictionary.

A simple test case is implemented in the next section where a broker uses SOAPpy and LinuxTuples to synchronize the communication between a client and a Web service through a tuple-space.

4.4.4 Implementation

A broker is implemented in this section using the SOAPpy toolkit. Figure 4.3 presents a scenario where a student indirectly requests services from a Web service through a tuple-space. The broker mediates the communication between both parties. The example is kept simple in order to identify how a broker is implemented using SOAPpy. Figure 4.4 shows a snapshot of the broker’s code. The implementation details are discussed by going through each step of the communication between the four parties:

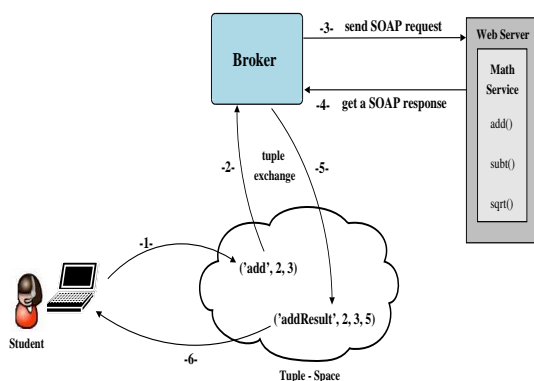


Figure 4.3: Using SOAPpy and LinuxTuples, the “Broker” is able to coordinate the communication between the student and the Math-Service through the tuple-space.

A snapshot of the Broker's code

```

001 #!/usr/bin/env python
002
003 import linuxtuples
004
005 # import the WSDL parser of SOAPpy
006 from SOAPpy import WSDL
007
008 # use proxy class 'WSDL.Proxy' to load
009 # the WSDL file and parse it.
010 service = WSDL.Proxy('http://...../Math/wsrf/math.wsdl')
011
012 # create a connection object to the tuple-server
013 conn = linuxtuples.connect(tuple_space, port)
014
015 # get a request from the tuple-space
016 tuple = conn.get(('add', None, None))
017
018 operation = tuple[0]
019
020 # Check the operation that the client is requesting
021 if operation == 'add':
022
023 # add request has been received. Invoke 'add' operation
024 # through the proxy object, with parameters tuple[1] and tuple[2].
025 result = service.add(tuple[1], tuple[2])
026
027 # the result retrieved from the SOAP response of the service
028 # is now in variable "result". put it in the space
029 conn.put(('addRes', tuple[1], tuple[2], result))
030
031 # check for other requests (i.e. subt(), etc.)
032 -----
  
```

Figure 4.4: A snapshot of the Broker’s code that is using both SOAPpy and LinuxTuples interfaces.

1. The student puts tuple $(\text{'add'}, 2, 3)$ into the space to request the addition of 2 and 3.

2. The broker receives the request. See the match between tuple (*'add', 2, 3*) and template (*'add', None, None*) at line *016* in the code.
3. The broker must make a SOAP request document from the parameters in tuple (*'add', 2, 3*) and sends the request to the service. To do this, the broker uses proxy class *WSDL.Proxy* offered by SOAPpy. The class creates a proxy object *service* that understands the service using its WSDL file (see line *010*). At line *025* the broker uses the proxy object *service* to invoke remote operation *add()*. It seems a local Python invocation, but in fact, XML processing and communications with the server take place behind the scene.
4. The service has finished adding 2 and 3. It sends the result in a SOAP response document to the broker. The *service* proxy object handles the SOAP response and maps the result from its XML representation to the corresponding Python representation. It is all abstracted from the developer and internally processed by the *service* object. The result is assigned to variable *result* at line *025*.
5. At line *029* the broker forms the necessary tuple *result* and uses operation *put()* of LinuxTuples to put the tuple into the space.
6. Tuple (*'addRes', 2, 3, 5*) holds the final result and it is in the tuple-space. The student can access it through standard LinuxTuples operations.

4.4.5 Results

Experiments were performed to identify the performance of the model presented in this chapter. The experiments compare two different scenarios:

1. The client directly invokes the service (*Full WS scenario* in Figure 4.1). The client is located on one node of the HP-Graphics cluster at SARA, described in section 3.4.1. The Web service is running on *torn.science.uva.nl* at the Section Computational Science (SCS) visualization lab.
2. The client uses the Web service through a tuple-space and the communication is synchronized by a broker (*Broker + Tuple-Space + WS* in Figure 4.3). The tuple-space and the Web service are on *torn.science.uva.nl*. The client and the broker are running on one node of the HP-Graphics cluster at SARA.

Two examples are used:

1. The *Math-Service* example of the previous section:

The time is measured between the moment the client requests the addition of two numbers and the moment the response is received by the client. Results are presented in table 4.1:

Table 4.1: *The Broker's overhead using the Math-Service example.*

	Full WS Scenario	Broker + Tuple-Space + WS
Response time	0.0189 sec	0.02 sec
Standard dev	0.002	0.001

The data size exchanged between the different components is small in this example. The overhead of the broker is negligible.

2. A *file transfer* example:

The client requests to download a file (7 MB) from a Web server. The time is measured between the moment the client initiates the request and the moment the file is received by the client. Results are presented in table 4.2:

Table 4.2: *The Broker's overhead using the file-transfer example.*

	Full WS Scenario	Broker + Tuple-Space + WS
Response time	2.7 sec	4.2 sec
Standard dev	0.1	0.18

In this example, the overhead of the broker is $4.2 - 2.7 = 1.5$ seconds.

Since communication with a Web service (exchange of XML) is slower than the communication with a tuple-space (exchange of binary code), the overhead can be reduced by placing the broker closer to the Web service. For this example, an experiment was performed where the broker is placed on the same machine with the Web service. The overhead is reduced to 0.45 seconds.

4.4.6 Discussion

A bridge model that integrates Web services and Linda systems was studied in this chapter. The model uses a broker that synchronizes the communication between both technologies. The broker is implemented using a WS-toolkit that offers common Web services functionality. A Python toolkit (SOAPpy) was adopted for testing purposes. Different experiments were performed with SOAPpy and results are summarized as follows:

- The toolkit supports a single programming language interface which is Python.
- The toolkit offers a high level of abstraction that shields developers from complicated XML/SOAP tasks.
- It is not possible to retrieve attachments with SOAPpy. The toolkit does not support complex data types as well, like multidimensional arrays for instance.
- Many of the WS -* family of specifications are not supported in SOAPpy. An experiment was performed to test the support for WS-Addressing. The result was negative.
- SOAPpy uses the Remote Procedure Call (RPC) binding style between the Web service and the SOAP messaging protocol. Currently, the trend is gradually shifting toward using document style binding.

SOAPpy is being integrated into the Zolera SOAP Infrastructure (ZSI) [50]. A library with more features for Python Web service. ZSI was not used in this chapter as it is still premature.

Alternative toolkits that support complex Web services scenarios are required. Potential implementations include the Java WS core [51]. This is a Java implementation of WSRF and a part of the Globus Toolkit 4 (GT4) [52]. It supports most of the requirements listed in section 4.4.2. pyGridWare is a Python implementation of WSRF that is compatible with Java WS core [53]. At the time of this writing, pyGridWare was still under development and needed more time to provide a stable release with support of different requirements. It could have been used in this chapter instead of SOAPpy, but its

documentation is poor and its installation procedure is complicated. Another toolkit for Web services is gSOAP; a cross platform toolkit that offers an XML to C/C++ language binding to ease the development of SOAP/XML Web services in C and C/C++ [54]. It is an open source project supervised by Robert van Engelen from Florida State University and Genivia, Inc. gSOAP supports most of the requirements listed above except for WSRF and GT4.

Different examples were implemented in order to test the model proposed in this chapter. The following remarks are concluded:

- Hard-coding a broker using a WS-toolkit as proposed in this model offers control over the broker. That is, it provides the ability to customize it depending on the application logic. This is important when using Linda especially in the context of complicated applications. Tuples that represent the results of a Web service invocation must be formatted by the broker using Linda's matching mechanisms in order to satisfy the application requirements and to discriminate between clients or processes that are using the same Web service. This is possible when the developer has direct control over the broker.
- Not all the implementations of Linda support the same data-types. For example, LinuxTuples supports only primitive data-types such as *integers*, *doubles*, *strings*, *etc.*. Therefore, a programmed broker can be customized to cope with the limitations of the underlying architecture.
- The model offers mobility to the broker. In some cases, a mobile broker helps reduce the additional overhead. Experiments in section 4.4.5 shows that placing the broker closer to the Web service decreased the overhead from 1.5 to 0.45 *seconds*. On the other hand, if the overhead of deserializing SOAP messages is higher than the communication overhead, the broker can be placed on the more powerful of the participating machines to decrease parsing and computational time.
- The model introduces the overhead of programming an additional component: the broker itself. The advantages discussed in the previous points must always dominate the overhead of developing the broker. For that reason, the underlying WS-toolkit must be easy and clear to develop with, and must provide a high level of abstraction for developers as in the case of SOAPpy, but with additional features.

The alternative approach is to generate the broker automatically from the WSDL interface of the Web service. This is worth to study as it reduces the overhead of developing the broker. Many challenges must be addressed in this case:

- The agreement between the clients and the broker on the format of the expected tuples. If the broker is generated in such a way that it expects the first parameter in a tuple to be the operation name, developers must be restricted to that rule otherwise no match will occur. The alternative is to specify keywords beside the parameters such as: (*'p1:2'*, *'p2:3'*, *'op:add'*). This requires the underlying Linda implementation to be reconfigured to adapt to this rule in order to prevent run-time errors.
- To customize the broker in order to cope with the application logic. When applications become complicated, complex matching mechanisms are required. This is not possible when the developer has no control over the Web service from which the broker is generated.
- If the Web service replies with a data-type that is not supported by the underlying Linda implementation, how would that be solved at run-time?

4.5 Conclusion

The model proposed in this chapter (Figure 4.2) integrates Web services and Linda. However, the followed approach for implementing the model (i.e. hard-coding the broker using a WS-toolkit) is not a complete solution. This thesis does not claim to have provided a complete solution for this problem and neither have set a complete solution as a goal target. The topic of this chapter is addressed by few scientific research in different contexts. Therefore, the purpose of addressing this question in this thesis was to investigate the underlying problems, to define requirements, to perform experiments and acquire the basic knowledge for future work.

Chapter 5

Prototype Use Cases

5.1 Introduction

In chapter 2 we discussed the requirements of IDSV supporting software and selected LinuxTuples as a suitable candidate. In this chapter we put this choice to the test by implementing two prototype applications. Each of these applications challenges different features in LinuxTuples which allows us to (1) test LinuxTuples' supported features and (2) to measure its performance under realistic scenarios.

The first application is a *simulated vascular reconstruction* that simulates arterial blood flow under different conditions. This application helps surgeons make treatment choices in the case of surgical vascular reconstruction *a priori*. The second is a *collaborative rigid body dynamics* application that provides a collaborative environment for multiple distributed end-users to experiment with rigid bodies. For each application, challenges are discussed in addition to the implementation approach. Results are presented and the features of LinuxTuples in IDSV environments are evaluated in the context of each application.

5.2 Simulated vascular reconstruction

Cardiovascular disease is still the major cause of death in the western world. The World Health Organization (WHO) estimates that for 2005; 59% of all deaths in women in Europe were due to cardiovascular disease and 45% of all deaths in men [55]. These are often caused either as a result of a *stenosis*; a narrowing in a blood vessel due to a buildup of plaque on the arterial walls, or as a result of an *aortic aneurysm*; a bulge in the aorta (the main artery coming out of the heart). Traditional ways of detection and diagnosis of such anomalies include X-ray Angiography, Computed Tomography Angiography (CTA) and Magnetic Resonance Angiography (MRA).

The treatment of cardiovascular disease depends on many factors. Usually there are several treatment choices, but which of these is optimal is not always obvious. The *simulated vascular reconstruction* environment developed at the University of Amsterdam can help a surgeon in making this choice [1, 2]. It calculates pressure, velocity and shear stress of blood flowing through the artery and simulates different conditions of the arteries caused by the characteristics of the blood flow [56, 57]. The application provides a visualization of the blood flow simulation and a mean of treatment simulation and/or surgical planning *a priori*.

5.2.1 Structure of the application

The application is composed of three main components: (1) the *blood flow simulation*, (2) a *concatenator* and (3) a *flow visualization*. Figure 5.1 shows a diagram that represents the structure of the application:

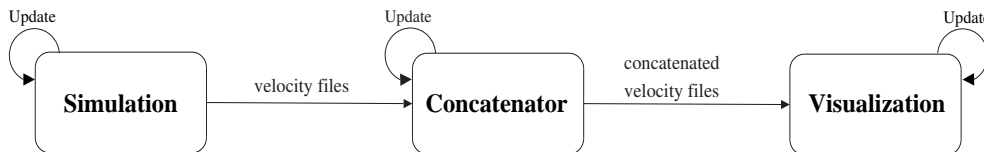


Figure 5.1: *The three components of the simulated vascular reconstruction application.*

The simulation was developed in the Section Computational Science (SCS) at the University of Amsterdam. It is based on the lattice Boltzmann BGK (LBGK) method [58]. It simulates steady and pulsatile unsteady blood flow in a model of the human abdominal aorta bifurcation. When executed on multiple processors (i.e. on a cluster of workstations), each processor participates in the simulation. The output of the simulation contains velocity vectors.

The concatenator groups all data that belong to iteration i (generated by each processor) into one velocity output file. The final output files are rendered by the visualization component developed using the visualization toolkit (VTK) [59].

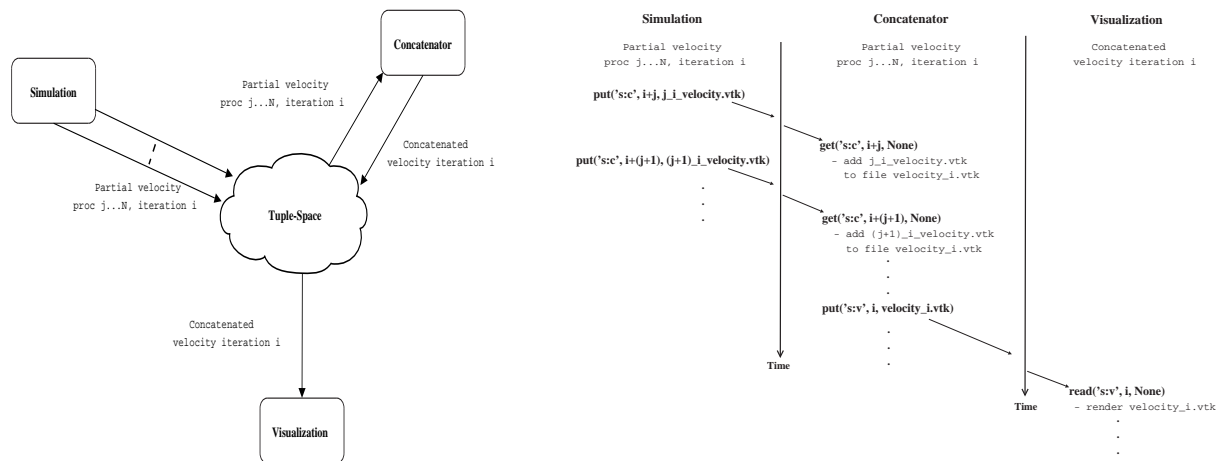
In the next section the application is modified into a distributed version where LinuxTuples is used as the communication and coordination layer between the components. This distributed version allows for a parallel execution of the components and offers a “live” representation of simulation results, in addition to providing the ability for end-users to interact with the simulation and/or the visualization. This is an IDSV application that challenges several of LinuxTuples’ features and performance.

5.2.2 Implementation

The application challenges LinuxTuples’ features in the following ways:

- The components of the application must be able to run on geographically distributed locations.
- The application must be able to show results of the simulation while the components are running in parallel (i.e. a pipelined execution). LinuxTuples must ensure “live” communication between the components.
- The components run on different system architectures and are implemented in different programming languages. The data representation must be unified and commonly interpreted by each participant component.
- The amount of data communicated between the components is substantial. LinuxTuples must provide a high throughput data transfer. Moreover, this is an IDSV application where responsiveness must be ensured in order to provide a comprehensive and interactive environment to the end-users.

The scenario of the application using LinuxTuples is presented in Figure 5.2(a). The components communicate through LinuxTuples and run simultaneously. Simulation data is represented to end-users as soon as results of the first iteration is available in the tuple-space. Figure 5.2(b) shows the sequence of events that take place between the three components. These are running in parallel in a *pipelined execution* strategy that was discussed in section 2.2.2. The Figure shows the tuples exchanged between the components as well.



(a) The execution scenario of the simulated vascular reconstruction application.

(b) A time diagram that shows the sequence of events and the exchanged tuples between the components.

Figure 5.2: *The execution scenario and a time diagram for the sequence of events.*

We go through each step of the execution scenario to identify how the components of this application interact using the tuple-space:

1. The simulation iteratively produces intermediate flow solutions. Each processor produces a tuple that contains a velocity data file. The group of data must be concatenated to form the complete result of this iteration.
2. The concatenator component takes the data generated by each processor for iteration i and adds it to file *velocity_i.vtk*. It is important to note that the concatenator does not wait for all processors to finish computing results of a given iteration. It starts concatenating the available data generated by processors that have already finished from that iteration and continues when new data is available.
3. Once the concatenator has finished grouping all data of iteration number i , it puts the result file *velocity_i.vtk* into the tuple-space.
4. The visualization component reads the tuple holding file *velocity_i.vtk* and renders it. At the same time the simulation and the concatenator components are running and generating new results that are put into the tuple-space.

The modification of the *simulated vascular reconstruction* application required approximately 50 lines of code added to the 4500 lines of the original simulation code. Each of the three components is written in a different programming language and runs on a different system architecture. For instance, the simulation is written in C++ and parallelized using the Message Passing Interface (MPI). The concatenator is also a C++ component that runs on any Linux machine, while the visualization is implemented in Python. The application is executed the same way as the original version with no changes at all, except that a tuple-space server has to run on a given host.

5.2.3 Results

An important question in this application is the performance of the resulting environment. This environment is required to be responsive and must provide comprehensive results to end-users. The components

exchange substantial amount of data that must be presented in parallel while the application is running. LinuxTuples is challenged in this application to ensure high data transfer throughput and high update rates. Moreover, the end-user must perceive simulation results at acceptable frame rates. For that reason, experiments with the application were performed in order to determine some performance figures like the update time, the update frequency and the data transfer rate.

In this application, the update time T_U is defined as:

$$T_U = T_{sim} + T_{sim \rightarrow concat} + T_{concat} + T_{concat \rightarrow vis} + T_{vis} \quad (5.1)$$

T_{vis} is the computational time of both the visualization and the renderer because these are grouped in the same process for this particular case. The response time T_R is not measured because an interactor component was not implemented.

In this experiment, the simulation sends data of one time-step to the tuple-space. The concatenator removes the data, groups it into one file and puts that file into the tuple-space. The visualization component reads the file from the tuple-space and represents it to the end-user. The time is measured from the moment the simulation starts sending data to the tuple-space until the moment the data is rendered. All experiments were performed using the SARA HP-Graphics cluster with configurations shown in table 5.1:

Table 5.1: *cluster characteristics and LinuxTuples throughput on the cluster.*

Number of nodes	29	
Processors	Intel Xeon 3.4 GHz	
Memory	2 GB	
Network	Network Card	Intel Gigabit NIC
	MTU	9000 bytes
	Latency	0.2 ms
LinuxTuples throughput	put()	156 MB/s
	get()	93 MB/s
	read()	93 MB/s

All nodes are connected to a Cisco Catalyst 4948 switch. This switch has a 96 Gb/s nonblocking switch fabric. Also, it has 2 ports of 10 Gb/s for external connectivity. Moreover, all nodes have a Gb/s full duplex connectivity to all other nodes.

Measurement results are presented in table 5.2:

Table 5.2: *Update time and update frequency for the simulated vascular reconstruction use case.*

	T_U	Variance	Standard Dev	$f_U = 1/T_U$
First frame	3.15 sec	0.2	0.4	0.3 Hz
Subsequent frames	2.30 sec	0.003	0.05	0.4 Hz

Different initializations take place for rendering the first frame. Therefore, a distinction is made between the first rendered frame and the subsequent ones.

The size of each transferred frame from one component to the other is 7 MB. Using table 5.1, we can see that each frame needs around 0.1 seconds to be transferred from one component to the other. Therefore, the data transfer rate in this application is 10 frames/sec. This is an upper limit to the application caused by the communication. The computational time of each component is also measured. Results are shown in table 5.3:

Table 5.3: *Computational time of each component.*

T_{sim}	0.09 sec
T_{vis}	1.02 sec
T_{concat}	1.0 sec

It is important to note that T_{sim} is measured when the simulation has already reached its optimum and therefore, it is producing iterative results of the last period.

In order to verify the correctness of the model in equation 5.1, we fill each variable with the obtained results of table 5.2 and table 5.3:

$$T_U = T_{sim} + T_{sim \rightarrow concat} + T_{concat} + T_{concat \rightarrow vis} + T_{vis}$$

$$2.30 \neq 0.09 + 0.1 + 1.0 + 0.1 + 1.02 = 2.31$$

The difference is $2.31 - 2.30 = 0.01$. This error is acceptable to confirm that the model presented in equation 5.1 is correct.

5.2.4 Discussion

With LinuxTuples, the application was distributed where each component was able to run on a different location. The components were also running in parallel in a pipelined execution strategy while simulation results were represented “live” to the end-user.

Components were executed on different system architectures and were written in different programming languages. LinuxTuples ensured a successful exchange of computational data through tuples. The components agreed on the form of these tuples (the matching mechanisms) and commonly interpreted their contents.

A model for the update time T_U of the application was introduced and verified through several performance measurements. These measurements showed that 0.1 seconds are required for two components to communicate on the underlying network. This can be considered as an overhead in respect to the original version of the application that did not offer any distribution. This overhead is acceptable only if the distribution of the components decreases the computational time by running each on a dedicated hardware. An update time $T_U = 2.30 \text{ sec}$ is acceptable for a “live” representation of simulation results. On the other hand, this result cannot be acceptable for interactivity. From that perspective, results of T_{vis} and T_{concat} lead us for two suggestions in order to increase the update time of the application:

- Using optimized algorithms for each component and running these on dedicated resources.
- Reducing the amount of transferred data in order to further increase the throughput of LinuxTuples. The work in [1] talks about two different techniques: (1) data encoding and (2) data compression. Both methods must be used with care as they introduce additional computational time for compression and decompression.

5.3 Collaborative rigid body dynamics

Rigid body dynamics studies the motions of bodies and their reactions when put under different forces in different environments. It is an interesting field for at least two applications: (1) *robotics* and (2) *video games*. In the first, it is important to understand how certain bodies will react to a robots' actions in order to successfully instruct the robot how to accomplish some tasks. In gaming, rigid body dynamics is used for improving realism.

Rigid bodies are composed of particles. Simulating the motion of a rigid body is similar to simulating the motion of a particle. Similar equations are used for both systems with more simplicity in the case of rigid bodies.

In this section, a LinuxTuples implementation of a *collaborative rigid body dynamics* is presented as an example of an IDSV application as well as an example of collaboration between geographically distributed end-users. This completes the circle of Figure 1.1 in chapter 1 and further explores the capabilities of LinuxTuples in IDSV environments. The application was developed by Bram Stolk from SARA Computing and Networking Services in Amsterdam [17].

5.3.1 Structure of the application

The application is composed of two main components:

1. A *simulation* that uses PyODE [60] to do rigid body dynamics. PyODE provides open source Python bindings for the Open Dynamic Engine - ODE [61]. This is a free library for simulating articulated rigid body dynamics.
2. An *interactor* that does both visualization (rendering) and allows users to interact with bodies in the scene by applying forces using a mouse input device. It uses PyOpenGL [62], which is a Python binding to OpenGL [63].

The interactor reads bodies and their state to draw the initial scene. Users interact with the bodies by applying forces. The simulation continuously reads the bodies and applies forces on the ones the user interacts with. All the components communicate and exchange messages through a tuple-space that synchronizes the activities of all participants.

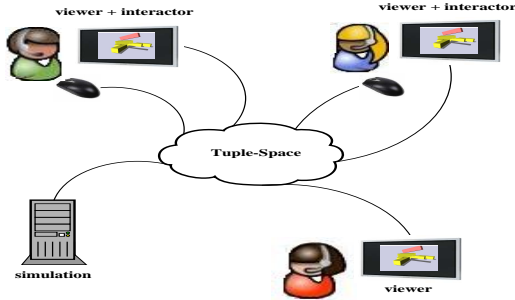
5.3.2 Implementation

The application imposes the following challenges on LinuxTuples:

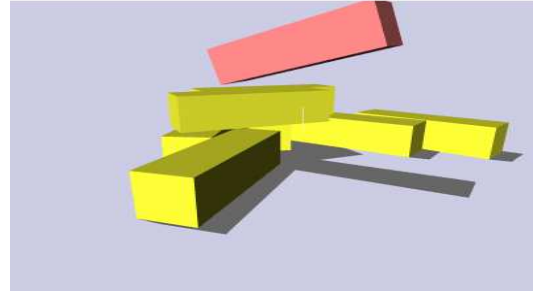
- Interaction must be implemented. Geographically distributed users must be able to collaborate through the tuple-space and interact with bodies.
- LinuxTuples must be able to synchronize between different participants and discriminate between their activities.
- Users must be able to join the application at run-time and collaborate with existing users.
- Interaction events must be received by the simulation with minimal delays to provide a useful collaborative environment for end-users.
- A user interaction with a body must be perceived in real-time. Otherwise the environment becomes frustrating and difficult to use.

5.3 Collaborative rigid body dynamics

The tuple-space synchronizes the activities of all participant components. The users collaborate through that space as shown in Figure 5.3(a).



(a) Two users interacting with rigid bodies through a tuple-space using a mouse. The third user is watching.

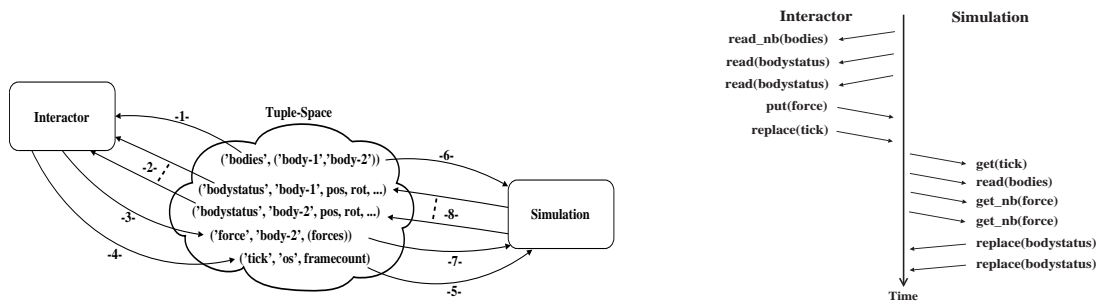


(b) A screen shot of the interactor application.

Figure 5.3: A collaborative rigid body dynamics scenario. The tuple-space synchronizes the simulation running on a separate server and the interactor application that is running at each user's host.

For each body, a *bodystatus* tuple of the form (*'bodystatus', 'bodyName', position, rotation...*) exists in the tuple-space to determine the state of the body. Forces applied on bodies by the interactor are represented in tuples of the form (*'force', 'bodyName', (forces)*). These *force* tuples exist in the tuple-space only when interactions take place.

TICK tuples are used to adapt the simulation to the interactor's pace. Each time a user interacts with a body, a TICK tuple is put into the tuple-space by the interactor and read by the simulation. This is important in order to prevent both components from falling apart in time or at least to reduce the time difference between them to the minimal. The interactor's time is the wall-clock while the simulation works on time-steps. If the interactor needs to move one body from position X to position Y in time *t*, the simulation must adapt and keep up with that speed. Otherwise, the simulation falls way behind the interactor and forces become higher. In consequence, the body explodes.



(a) An execution scenario of the collaborative rigid body dynamics application.

(b) A time diagram that shows the sequence of events.

Figure 5.4: An execution scenario and a time diagram for the sequence of events.

Adapting both components also helps to face the consequences of the network latency. Latency issues related to this application are discussed in more details in the next section.

Figure 5.4 presents an execution scenario and the sequence of events that take place when two bodies are used. We go through each step to identify how the components of this application interact using the tuple-space:

1. The interactor reads the bodies that exist in the tuple-space.
2. For each body, the interactor reads its *bodystatus* tuple in order to draw it.
3. A *force* tuple is put into the tuple-space when a user interacts with a body. In this scenario, the user interacts with 'body-2'.
4. A TICK tuple is put into the tuple-space by the interactor. This is to adapt the simulation to the interactor's pace.
5. The simulation removes the TICK tuple from the space.
6. The simulation reads the bodies that exist in the space.
7. For each body, the simulation tries to get forces applied on that body. Note the use of the non-blocking operation *get_nb()*. This is important because if no forces were applied on a certain body, the simulation can skip to the next body without blocking.
8. After applying forces, the simulation "replaces" all existing *bodystatus* tuples with updated ones even for bodies that were not interacted with.

It is very important to note the use of a *replace()* operation as represented in the time diagram of Figure 5.4(b). This operation is new to LinuxTuples and was added by Bram Stolk for the purpose of this application. Two versions of *replace()* exist:

- **replace(*template*, *t*)**. Replaces the first tuple in the space that matches the *template* with a tuple *t*. Another use of this operation is through **replace(*t*₁, *t*₂)** that replaces a specific tuple *t*₁ with tuple *t*₂.
- **replace_be(*template*, *t*)** or **replace_be(*t*₁, *t*₂)**. This is the *best effort* version of the operation.

Everytime an interaction takes place, all participant interactors must read updated *bodystatus* tuples in order to draw the new resulting scene. Therefore, old *bodystatus* tuples must not remain into the tuple-space. Otherwise, interactors might read them which causes incoherent results (i.e. new positions of some bodies will not be drawn). The only way in LinuxTuples to put data into the space is through operation *put()* that in this case, causes old *bodystatus* tuples to remain. The alternative is to remove each old *bodystatus* tuple using *get()* and replace it using operation *put()*. This introduces communication overhead for using two operations *get()/put()* for each *bodystatus* tuple. This motivated the implementation of operation *replace()* that ensures an atomic read-modify-write transaction.

5.3.3 Results

Different types of experiments were performed in order to determine the performance of LinuxTuples in this IDSV application. Two major concerns are concluded from these experiments: (1) the effects of the network latency on the responsiveness of the application and (2) the implementation approach.

Saving each state of a body in a separate tuple is an elegant approach. However, it makes the number of communications twice the number of the bodies in the space. For instance, after each interaction

the simulation puts a *bodystatus* tuple into the space for each existing body. The interactor reads all these *bodystatus* tuples in order to draw the new scene. Therefore, when the number of bodies increases, the effects of the network latency become more apparent on LinuxTuples communications (TCP/IP per communication) and the responsiveness of the application drastically decreases.

The latency of the underlying network and the behavior of LinuxTuples with respect to this latency has a negative effect on the usefulness of the application even with small numbers of blocks. When the latency increases, interactions with the bodies take longer time to be received by the simulation. In consequence, the interactor and the simulation fall apart in time. In other words, the difference in time between the position of the simulation and the position of the interactor becomes larger. This leads to bigger forces and causes the explosion of the body. Using TICK tuples to adapt the simulation to the interactor's pace helps get away with the latency. However, this adaptation does not work if the latency is higher than a certain threshold value. An experiment was performed on two different networks each with a different latency:

1. Between SARA and the University of Amsterdam where the latency is 0.6 ms. The performance was acceptable and the application was useful for real-time interaction through the tuple-space.
2. Between SARA and a node outside of the city with a 22 ms network latency. The performance severely decreased to the point of making the application unusable.

According to these results, if the implementation is optimized to eliminate the negative effects of adding new blocks, the application can be used on a Local Area Network for an increasing number of blocks with acceptable responsiveness and real-time interaction.

5.3.4 Discussion

Interaction was successfully implemented with LinuxTuples. Geographically distributed users were able to collaborate through the tuple-space and interact with the simulation component by moving rigid bodies through a visualization/renderer component.

LinuxTuples synchronized the activities of all participants. New users were able to join at run-time and collaborate with existing end-users without altering the execution of the application. This was possible due to the space/time distribution offered by LinuxTuples.

Implementation wise; the application can be optimized. For instance, the state of all existing bodies can be saved in the same tuple instead of using separate tuples for each body. This reduces the communication overhead and allows the use of the application with more bodies. Using threads to simultaneously read *bodystatus* tuples can be an alternative as well.

A new operation (*replace()*) was added to LinuxTuples as the application revealed a scenario for which the original interface lacked the required service. This operation ensures the coherency of the data in the tuple-space when atomic read-modify-write transactions are required and reduces the communication overhead that result from using both *get()/put()* operations for the same scenario.

The major concern in this application is the performance of LinuxTuples. Extensively opening TCP/IP connections makes LinuxTuples vulnerable to the latency of the underlying network. Experiments with the application showed that the responsiveness is acceptable on Local Area Networks using small numbers of bodies. On the other hand, the application was unusable on wider networks with high latencies.

5.4 Conclusion

Two IDSV applications were implemented using LinuxTuples. These applications challenged different features in the architecture and helped draw the following concluding remarks:

- It is possible to build IDSV applications using LinuxTuples.
- The interface of the architecture is easy to use and clear as well. However, it does not cover all IDSV possible scenarios. Extra work is required from the developer when scenarios become complicated. One such case motivated the addition of a new operation *replace()* to provide an easy solution to a required service. LinuxTuples is open source, so it can be modified and extended.
- LinuxTuples distributes components in space and time. The space distribution allows components to run on dedicated resources over geographic areas. The time distribution offers the flexibility that IDSV applications require. That is, new components or users can join/leave the environment at run-time with minimal definitions and without explicit communications between components.
- LinuxTuples allows components to run in a *pipelined execution* strategy. Components of the *simulated vascular reconstruction* application were able to run in parallel, offering “live” representation of simulation results.
- Two characteristics impose a major concern on the performance of LinuxTuples: (1) the centralized tuple-space and (2) the extensive use of TCP/IP sockets to communicate with the tuple-space. A centralized tuple-space causes components to compete for the same resource (i.e. memory) which might get depleted. Extensively using TCP connections introduces the overhead of opening/closing a connection and exhausts the socket address space. This had major consequences on the responsiveness of the *collaborative rigid body dynamics* application and restricted it to run on Local Area Networks.

Chapter 6

Summary and Future Directions

6.1 Summary

The rapid advance in performance and capabilities of computational resources is offering scientists the opportunity to explore complex phenomena through computer simulations. These simulations generate large and complex data that is hard to analyse with numerical algorithms, either because these algorithms are difficult to design due to the complexity of the data or because they take too long to produce acceptable results. The alternative is to represent the data using scientific visualization that offers more insight into the underlying problem, especially through advanced display devices and virtual reality environments that are available to provide comprehensive and accurate representations.

In addition to generating large data volumes and complex results, computer simulations are often intractable. That is, their end-result can be obtained only through explicit execution from start to end without short cuts. These results might take too long before they are computed and it is often the case that many explicit executions are required before reaching an optimal result. Human intervention through interactive simulations helps reduce this computational time by altering the course of the simulation and changing its direction at run-time. Human interaction with the visualization is also important. It makes it possible for end-users to interact with the visual representation in order to obtain different views and more insight into the underlying data set. The complexity of the components (i.e. simulations, visualizations and renderers) involved in this type of applications raises the need to run each on dedicated resources to increase the performance and reduce the processing time of the system as a whole. The available network bandwidth (which doubles every 9 months) encourages the distribution of these components on dedicated hosts. The result is an *Interactive Distributed Simulation and Visualization* (IDSV) that offers a collaborative problem solving environment for complex and real-time problems.

This thesis studies the design, implementation and use of an architecture that offers communication and coordination between geographically distributed IDSV components. The architecture is challenged by different requirements. Chapter 2 discusses these requirements and compares between existing communication architectures. In IDSVs the number of participants is non-deterministic. Components or users can join/leave the environment at run-time while the application is running. Interaction events are non-deterministic as well. It is not defined when an interaction is going to take place and who is involved in a certain interaction. Participant components are geographically distributed which means they must communicate and exchange data. This introduces a communication overhead that must be minimal to fully exploit the advantage of running components on distributed dedicated hardware. The data transfer rate must be high and delays must be reduced to offer a responsive and comprehensive environment to the end-user. Moreover, in IDSVs multiple users from different institutes might collabo-

rate to solve common problems. As a consequence, the components and their output are heterogeneous. Based on its capabilities in respect with these scenarios, LinuxTuples was selected as a suitable communication and coordination architecture for IDSV applications. Chapter 3 introduces LinuxTuples and studies its different features. LinuxTuples is an implementation of the Linda coordination model. It distributes processes in space and time using a tuple-space (i.e. blackboard-like space) as a distributed shared memory. This offers flexible communication between participant processes that use tuples to exchange data (i.e. computational requests, computational results and messages). Despite that many of the required mechanisms can be implemented in LinuxTuples, the architecture lacks crucial facilities like garbage collection, failure recovery and transaction roll-back. The absence of these facilities obliges developers to do extra work to implement complicated scenarios.

In multidisciplinary collaborative problem solving environments, computational components originate from different technologies. Web services are one of these technologies that are increasingly gaining interest in the Web and distributed computing communities. They offer the possibility to develop heterogeneous, Internet scale and loosely coupled applications using open protocols and standards. Chapter 4 studies a bridge model to integrate Web services and Linda systems. The research identifies the differences between both technologies, the basic facilities the bridge must implement and the requirements for a successful integration. A prototype was implemented and tested. The proposed model is not a final complete solution. The lessons learned and the experiments that were performed are the basic building blocks for future work.

LinuxTuples as the selected architecture for building IDSV applications was put under test in Chapter 5. Two IDSV prototype test cases were implemented: (1) a *simulated vascular reconstruction* and (2) a *collaborative rigid body dynamics*. Each application challenged different features in LinuxTuples. The first required a high transfer rate for substantial amounts of data and a *pipelined execution* strategy for its distributed components. The second required interactivity, synchronization of interaction events and low latency communication to offer a useful collaborative environment. Both applications were successfully implemented and LinuxTuples offered the required facilities. However, a major concern in LinuxTuples is its vulnerability to the underlying network latency. This is related to the way LinuxTuples operations communicate with the tuple-space. They extensively use TCP sockets which introduce overhead (i.e. opening/closing TCP connections). This restricts LinuxTuples to be used on Local Area Networks only.

6.2 Conclusion

We have chosen LinuxTuples in this thesis as an architecture that implements the coordination model of Linda. Results we obtained indicate that the concept of Linda is suitable for building IDSVs. LinuxTuples itself is a potential candidate. It successfully faces many of the requirements imposed by IDSV applications. We cannot yet claim that LinuxTuples is a complete solution capable to cover all IDSV scenarios. However, the architecture is open-source and different extensions have been added to it by Bram Stolk through the joint cooperation between the Scientific Visualization and Virtual Reality (SVVR) group and the Insight group of SARA Computing and Networking Services. Future extensions and enhancements for LinuxTuples are being studied within the SVVR group to make it a robust architecture for IDSV applications.

As for the integration of Web services and Linda, very few projects have investigated this question and for totally different contexts. They either focus on making the tuple-space a repository of XML documents or they implement Linda's interface within a Web service. The work in this thesis is the first to tackle the problem for IDSV environments. We focus on integrating both technologies while preserving their character. The proposed model gives control to developers to customize the bridge according to the underlying Linda implementation and to the underlying application. Despite that the final work is not complete, the results we obtained are considered as the basis for future research in this domain.

6.3 Future work

The research described in this thesis gives rise to new research questions that will need to be addressed in future projects. We conclude that LinuxTuples currently lacks some features and it suffers from network latency. Two alternatives are proposed to enhance the performance of LinuxTuples:

1. Investigate the possibility of modifying the way processes in LinuxTuples communicate with the tuple-space. For example, by opening one TCP connection per process instead of opening a TCP connection per operation.
2. Compare the performance of LinuxTuples with existing Linda implementations and identify the areas where these are better and borrow from them. For instance, pyLinda [11] uses multiple distributed tuple-spaces. If experiments show that this is more advantageous than a centralized LinuxTuples space, extend LinuxTuples to support distributed tuple-spaces.

As for the missing features in LinuxTuples, a protocol must be studied in order to support different scenarios through automatic mechanisms. For instance, the protocol must offer solutions for tuples management, tuple-space cleanup and failure recovery.

The *simulated vascular reconstruction* IDSV application was not fully exploited in this thesis. Interaction capabilities can be added to this application using LinuxTuples. This makes the application fully interactive and therefore, can be used for testing purposes. For example, to test added features to LinuxTuples.

The bridge model that integrates Web services and Linda offers room for many open questions. These can be summarized in the following points:

- Compare existing WS-toolkits in order to identify which one is most suitable to implement the broker. The toolkit must support requirements discussed in section 4.4.2. Moreover, the toolkit must offer the highest possible level of abstraction for developers.
- Investigate the possibility to access grid resources using the bridge model. The *simulated vascular reconstruction* application can be used as a test case. The following scenario can be implemented: the bridge mediates the communication between the simulation and grid services through a tuple-space in order for the simulation to access the grid.
- Address the approach of automatically generating a broker from the WSDL interface of the Web service. This is possible for tightly coupled applications. However, the challenge is in the context of loosely coupled IDSVs with complicated scenarios and where complex matching mechanisms are required.

Bibliography

- [1] R.G. Belleman. *Interactive Exploration in Virtual Environments*. PhD thesis, University of Amsterdam, April 2003.
- [2] R.G. Belleman and P.M.A. Sloot. The design of dynamic exploration environments for computational steering simulations. In Marian Bubak, Jacek Mościński, and Marian Noga, editors, *Proceedings of the SGI Users' Conference*, pages 57–74, Kraków, Poland, October 2000. Academic Computer Centre CYFRONET H. ISBN 83-902363-9-7.
- [3] Benno J. Overeinder and Peter M. A. Sloot. Extensions to time wrap parallel simulation for spatially decomposed applications. In David Al-Dabass and Russel Cheng editors, editors, *Proceedings of the fourth United Kingdom Simulation Society Conference (UKSim99)*, pages 67–73, Cambridge, UK, April 1999.
- [4] R.G. Belleman and R. Shulakov. High performance distributed simulation for interactive simulated vascular reconstruction. In P.M.A. Sloot, C.J. Kenneth Tan, Jack J. Dongarra, and Alfons G. Hoekstra, editors, *International Conference on Computational Science (ICCS), Amsterdam, the Netherlands*, volume 2331 of *Lecture Notes in Computer Science (LNCS)*, pages 265–274, Berlin, April 2002. Springer-Verlag. ISBN 3-540-43594-8.
- [5] J. Leigh, A. Johnson, and T. DeFanti. Cavern: A distributed architecture for supporting scalable persistence and interoperability in collaborative virtual environments. *Virtual Reality: Research, Development and Applications*, 2(2):217237, 1997.
- [6] Distributed Interactive Simulation. From Wikipedia, the free encyclopedia, February 14 2007. On the web: http://en.wikipedia.org/wiki/Distributed_Interactive_Simulation.
- [7] J. Mark Pullen and Vincent P. Laviano. A Selectively Reliable Transport Protocol For Distributed Interactive Simulation. *Center For Excellence in Command, Control, Communications and Intelligence. George Mason University*.
- [8] James Hardt and Kevin White. Distributed Interactive Simulation (DIS), February 14 2007. On the web: <http://www-ece.engr.ucf.edu/~jza/classes/4781/DIS/project.html>.
- [9] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 6(1):80–112, 1985.
- [10] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communication of the ACM*, 35(2), February 1992.
- [11] PyLinda, December 23 2006. On the web: <http://www-users.cs.york.ac.uk/~aw/pylinda/doc/index.html>.

BIBLIOGRAPHY

- [12] Z. Zhao, R.G. Belleman, G.D. van Albada, and P.M.A. Sloot. AG-IVE: an agent based solution to constructing interactive simulation systems. In P.M.A. Sloot, C.J. Kenneth Tan, Jack J. Dongarra, and Alfons G. Hoekstra, editors, *International Conference on Computational Science (ICCS), Amsterdam, the Netherlands*, volume 2329 of *Lecture Notes in Computer Science (LNCS)*, pages 693–703, Berlin, April 2002. Springer-Verlag. ISBN 3-540-43594-8.
- [13] E.V. Zudilova, P.M.A. Sloot, and R.G. Belleman. A multi-modal interface for an interactive simulated vascular reconstruction system. In *Fourth IEEE International Conference on Multimodal Interfaces (ICMI '02)*, pages 313–318, Pittsburgh, Pennsylvania, 14-16 October 2002. IEEE Computer Society.
- [14] Paul Dechering, Rix Groenboom, Edwin de Jong, and Jan Tijmen Udding. Formalization of a software architecture for embedded systems: a process algebra for splice. In *Proceedings of the 32nd Hawaii International Conference on System Sciences*, volume 3. IEEE Computer Society.
- [15] Defense modeling and simulation office (DMSO). High Level Architecture (HLA), December 19 2006. On the web: <http://hla.dmsomil/>.
- [16] Marco Brasse and Nico Kuijpers. Standardising distributed simulations: The High Level Architecture. *Xootic Magazine*, 7(1):1624, July 1999.
- [17] SARA: Computing and Networking Services, December 20 2006. On the web: <http://www.sara.nl>.
- [18] Universiteit van Amsterdam, Scientific Visualization and Virtual Reality Group, February 14 2007. On the web: <http://www.science.uva.nl/research/scs/visualization/>.
- [19] Getting started with JavaSpaces technology: Beyond conventional distributed programming paradigms, December 23 2006. On the web: <http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/index.html>.
- [20] JS - Javaspaces Service Specification, February 4 2007. On the web: <http://www.sun.com/software/jini/specs/jini1.2html/js-spec.html#27387>.
- [21] Make room for JavaSpaces, December 23 2006. On the web: <http://www.javaworld.com/jw-11-1999/jw-11-jiniology.html?page=1>.
- [22] TSpaces Intelligent Connectionware, December 23 2006. On the web: <http://www.almaden.ibm.com/cs/TSpaces/intro.html>.
- [23] The Lime: Linda in a Mobile Environment, December 23 2006. On the web: <http://lime.sourceforge.net/>.
- [24] Nigel Davies, Adrian Friday, Wade S., and Gordon Blair. An asynchronous distributed systems platform for heterogeneous environments. In *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*. Sintra, Portugal, 7-10 September.
- [25] A. Hawick, H.A. James, and L.H. Pritchard. Tuple-Space based middleware for distributed computing. Computer Science Division, School of Informatics. Bangor, North Wales, LL57 1UT, UK.
- [26] Ruple: An XML Space Implementation, December 23 2006. On the web: http://www.idealliance.org/papers/xml02/dx_xml02/papers/04-05-03/04-05-03.html.

- [27] Robert Tolksdorf and Dirk Glaubitz. XMLSpaces for coordination in Web-based systems. In *Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, page 322. wet ice, 2001.
- [28] George C. Wells. A tuple Space Web service for distributed programming. Department of Computer Science, Rhodes University. Grahamstown, 6140, South Africa.
- [29] Roberto Lucchi and Gianluigi Zavattaro. WSSecSpaces: a secure data-driven coordination service for web services applications. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 487–491, Nicosia, Cyprus, 2004. ACM Press.
- [30] The auction site eBay, March 16 2007. On the web: <http://www.ebay.com/>.
- [31] eBay developers program, March 16 2007. On the web: <http://developer.ebay.com/businessbenefits/faq/whatis ebaywebservices>.
- [32] Amazon Web Services, March 16 2007. On the web: <http://www.amazon.com/gp/browse.html?node=3435361>.
- [33] Google APIs, March 16 2007. On the web: <http://code.google.com/>.
- [34] Web Services Description Language (WSDL) 1.1, December 21 2006. On the web: <http://www.w3.org/TR/wsdl>.
- [35] Latest SOAP versions, December 21 2006. On the web: <http://www.w3.org/TR/soap/>.
- [36] W3C World Wide Web Consortium, February 28 2007. On the web: <http://www.w3.org/>.
- [37] Web Services Addressing (WS-Addressing), February 28 2007. On the web: <http://www.w3.org/Submission/ws-addressing/>.
- [38] Grid Computing Info Centre (GRID Infoware), May 5 2007. On the web: <http://www.gridcomputing.com/>.
- [39] TIME.WAIT sockets in Debian GNU/Linux, March 16 2007. On the web: <http://www.stolk.org/debian/timewait.html>.
- [40] Lmbench - Tools for performance analysis, January 18 2007. On the web: <http://www.bitmover.com/lmbench/>.
- [41] e-Science, March 28 2007. On the web: <http://en.wikipedia.org/wiki/E-Science>.
- [42] W3C, Web Services Activity, March 28 2007. On the web: <http://www.w3.org/2002/ws/>.
- [43] International Journal of Web and Grid Services (IJWGS), March 28 2007. On the web: <https://www.inderscience.com/browse/index.php?journalID=47>.
- [44] OASIS Web Services Resource Framework (WSRF) TC, March 28 2007. On the web: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.
- [45] W3C, Web Services Activity Statement, March 28 2007. On the web: <http://www.w3.org/2002/ws/Activity>.
- [46] The Globus Toolkit 4 programmer’s tutorial, December 21 2006. On the web: <http://www.casa-sotomayor.net/gt4-tutorial/>.

BIBLIOGRAPHY

- [47] Universal Description, Discovery and Integration, March 28 2007. On the web: <http://www.uddi.org/>.
- [48] The Apache Ant Project, May 5 2007. On the web: <http://ant.apache.org/>.
- [49] Introduction to WS-Addressing, February 22 2007. On the web: http://dev2dev.bea.com/pub/a/2005/01/ws_addressing_intro.html.
- [50] Pythong Web Services Project, February 26 2007. On the web: <http://pywebsvcs.sourceforge.net/>.
- [51] Java WS Core, March 16 2007. On the web: <http://www.globus.org/toolkit/docs/4.0/common/javawscore/>.
- [52] The Globus Alliance, December 19 2006. On the web: <http://www.globus.org/>.
- [53] Python WSRF Programmers' Tutorial, February 28 2007. On the web: <http://dsd.lbl.gov/gtg/projects/pyGridWare/doc/tutorial/html/index.html>.
- [54] gSOAP: C/C++ Web Services and Clients, February 28 2007. On the web: <http://www.cs.fsu.edu/~engelen/soap.html>.
- [55] Dr Catherine Le Gal-Camus. Address to the informal meeting of EU health ministers, January 3 2007. On the web: http://www.who.int/nmh/media/speeches/nmh_adg_speech_eu_april06.en.pdf.
- [56] A.M.M. Artoli, A.G. Hoekstra, and P.M.A. Sloot. Simulation of a systolic cycle in a realistic artery with the Lattice Boltzmann BGK Method. *International Journal of Modern Physics B*, 17(12):95–98, 2003.
- [57] A.M. Artoli, A.G. Hoekstra, and P.M.A. Sloot. Mesoscopic simulations of systolic flow in the human abdominal aorta. *Journal of Biomechanics*, 39(2006):873–884, January 26 2005.
- [58] SCS Lattice Boltzmann Research, December 20 2006. On the web: http://www.science.uva.nl/research/scs/projects/lbm_web/index.html.
- [59] The Visualization Toolkit (vtk), January 5 2007. On the web: <http://public.kitware.com/VTK/>.
- [60] PyODE, January 28 2007. On the web: <http://pyode.sourceforge.net/>.
- [61] Open Dynamic Engine, January 28 2007. On the web: <http://ode.org/>.
- [62] PyOpenGL - the Python OpenGL Binding, January 28 2007. On the web: <http://pyopengl.sourceforge.net/>.
- [63] OpenGL - Open Graphics Library, January 28 2007. On the web: <http://www.opengl.org/>.