TripCom

Triple Space Communication

FP6 – 027324

Deliverable

D6.5v2

Towards a Scalable Triple Space v2.0

Reto Krummenacher
Elena Simperl
Daniel Martin
Michael Lafite
Christian Schreiber
Alessio Carenini
David de Francisco

April 28, 2009

Executive Summary

The TripCom scalability task was defined as reaction to the feedback received during the first project review. This new effort is part of WP6 and is dedicated to the study of core principles at design, architecture and implementation level in order to ensure the realization of a Web-scale Triple Space. This deliverable is a follow-up to the M24 deliverable D6.5, and focuses on the last step of the TripCom scalability task: evaluate the final system with respect to its scalability, based on the given procedures and concepts. In order to do so, TripCom provided the necessary implementations to deploy the TripCom infrastructure on Amazon EC2, and defined a testing environment and test clients. Successful results, critical observations, and problems with the implementation that could be detected in the evaluation process are presented in this deliverable.

Document Information

IST Project Number	FP6 – 027324	**Acronym**	TripCom
Full Title	Triple Space Communication		
Project URL	http://www.tripcom.org/		
Document URL			
EU Project Officer	Werner Janusch		

Deliverable	**Number**	6.5v2	**Title**	Towards a Scalable Triple Space v2.0
Work Package	**Number**	6	**Title**	Triple Space Architecture and Component Integration

Date of Delivery	**Contractual**	M36	**Actual**	31-March-09
Status	draft		final ☐	
Nature	prototype ☐ report ☒ dissemination ☐			
Dissemination Level	public ☒ consortium ☐			

Authors (Partner)	UIBK			
Resp. Author	Reto Krummenacher		**E-mail**	reto.krummenacher@sti2.at
	Partner	Semantic Technology Institute, University Innsbruck)	**Phone**	+43 512 507 6452

Abstract (for dissemination)	This deliverable is dedicated to the evaluation and assessment of the core principles at design, architecture and implementation of a Web-scale Triple Space.

Keywords	Scalability, Deployment, Evaluation, Web-scale Triple Space

Version Log			
Issue Date	**Rev No.**	**Author**	**Change**
2009-02-27	1	Reto Krummenacher	Preliminary structure and content outline
2009-03-20	2	Christian Schreiber	Scalability test environment
2009-03-27	3	Reto Krummenacher	Beyond TripCom part
2009-03-30	4	Christian Schreiber	Scalability results
2009-03-30	5	Michael Lafite	Test environment and scalability results
2009-03-31	6	Alessio Carenini	Usecase evaluation eHealth
2009-03-31	7	David de Francisco	Usecase evaluation EAI
2009-04-01	8	Reto Krummenacher	Intro / Conclusion
2009-04-05	9	Reto Krummenacher	Update and finalization evaluation section
2009-04-12	10	Reto Krummenacher	Release review version
2009-04-28	11	Reto Krummenacher	Incorporation review comments and final release

Project Consortium Information

Acronym	Partner	Contact
Semantic Technology Institute Innsbruck `http://www.sti-innsbruck.at`	STI	Prof. Dr. Dieter Fensel Semantic Technology Institute (STI) Innsbruck, Austria E-mail: dieter.fensel@sti-innsbruck.at
National University of Ireland, Galway `http://www.deri.ie`	NUIG	Dr. Laurentiu Vasiliu Digital Enterprise Research Institute (DERI) Galway, Ireland Email: laurentiu.vasiliu@deri.org
University of Stuttgart `http://www.iaas.uni-stuttgart.de/`	USTUTT	Prof.Dr. Frank Leymann Inst. für Architektur von Anwendungssystemen (IAAS) Stuttgart, Germany E-mail: frank.leymann@informatik.uni-stuttgart.de
Vienna university of Technology `http://www.complang.tuwien.ac.at/`	TUW	Prof.Dr. eva Kühn Institut für Computersprachen Vienna, Austria E-mail: eva@complang.tuwien.ac.at
Free University Berlin `http://www.ag-nbi.de/`	FUB	Prof. Dr.-Ing. Robert Tolksdorf AG Netzbasierte Informationssysteme Berlin, Germany E-mail : tolk@inf.fu-berlin.de
Ontotext Lab, Sirma Group Corp. `http://www.ontotext.com/`	ONTO	Atanas Kiryakov, Vassil Momtchev, Ontotext Lab, Sirma Group Corp. Sofia, Bulgaria E-mail: vassil.momtchev@ontotext.com
Profium OY `http://www.profium.com/`	Profium	Dr. Janne Saarela Profium OY Espoo, Finland E-mail: janne.saarela@profium.com
CEFRIEL SCRL. `http://www.cefriel.it/`	CEFRIEL	Davide Cerri CEFRIEL SCRL. Milano, Italy E-mail: cerri@cefriel.it
Telefonica I+D `http://www.tid.es/`	TID	Noelia Pérez Crespo Telefonica I+D Madrid, España E-mail: npc@tid.es

Table of Contents

1 Introduction 2

2 Towards a Scalable Triple Space v1 - Revisited 4

3 Scalability Test Environment 9
- 3.1 Amazon EC2 . . . 9
  - 3.1.1 EC2 I/O and CPU Performance Guarantees . . . 9
  - 3.1.2 High-Level Architecture . . . 10
  - 3.1.3 Custom Tooling . . . 11
- 3.2 Test Environment . . . 11
  - 3.2.1 Scalability Clients . . . 11

4 Evaluation of the Prototype 15
- 4.1 Generic Scalability Benchmarks . . . 15
  - 4.1.1 Read Operation in Various Distributed Settings . . . 16
  - 4.1.2 Read Operation Under Varying Workload . . . 18
  - 4.1.3 Read Operation Without Space Indication . . . 20
  - 4.1.4 Publication via 'out' Operation . . . 22
  - 4.1.5 Multiset Operations of the Extended API . . . 22
  - 4.1.6 The Further Extended API . . . 23

5 Evaluation of the Use Cases 24
- 5.1 EAI . . . 24
  - 5.1.1 Marketplace Design Concerns . . . 24
  - 5.1.2 Marketplace Deployment Models . . . 25
  - 5.1.3 Recommendations . . . 26
- 5.2 eHealth . . . 27
  - 5.2.1 Patient Summary on a Single Kernel . . . 27
  - 5.2.2 Patient Summary on Multiple Kernels . . . 29

6 Scalability Issues Beyond TripCom 31

7 Conclusions 35

List of Abbreviations

AMI	Amazon Machine Image
API	Application Programming Interface
CAN	Content-Addressable Network
CMR	Complete-Model-then-Revise
DAM	Digital Asset Management
DBMS	Database Management System
DHT	Distributed Hash Table
DNS	Domain Name Server
DSB	Distributed Service Bus
EAI	Enterprize Application Integration
EC2	Elastic Compute Cloud
EPS	European Patient Summary
ESB	Enterprise Service Bus
EU	European Union
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
LarKC	Large Knowledge Collider
ORDI	Ontology Representation and Data Integration
OWL	Web Ontology Language
P2P	Peer-To-Peer
RDF	Resource Description Framework
RDFS	RDF Schema
SSH	Secure Shell
SPARQL	SPARQL Protocol and RDF Query Language
S3	Simple Storage Service
SOA	Service Oriented Architecture
TCP	Transmission Control Protocol
TripCom	Triple Space Communication
TS	Triple Space
TUW	Technische Universität Wien
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WP	Work Package
WS	Web Service
WSMO	Web Service Modeling Ontology
WSMX	Web Service Execution Environment
XML	Extensible Mark-up Language

1 Introduction

The World Wide Web is one of the ground-breaking achievements of modern computer science. The core technology, so simple to use, allows billions of users to publish, share and consume information from all over the world. With the emergence of the Web2.0 paradigm, Web services and the Semantic Web, the World Wide Web has transformed to an open interactive market place. Heterogeneous and distributed producers and consumers – or rather, in the current context of the Web, so-called prosumers; actors on the Web that interactively produce and consume content or functionality (services) – have to integrate data and to coordinate their activities. Bringing this all together and lifting it to the level of large-scale Web computing calls upon scalability, heterogeneity and dynamism challenges.

The TripCom project proposes in response to these challenges a semantics-aware tuplespaces as middleware for the Semantic Web and Semantic Web service. The TripCom infrastructure enables the integration, communication and coordination of billions of autonomous, distributed and heterogeneous data prosumers. In TripCom information and service sources are no longer addressed directly by their endpoint locator, but are abstracted by the Triple Space middleware that provides a unified and virtually global view of the data published in diverse databases. Such middleware is the fundamental building block for the realization of distributed application in the true Web sense: scalable, open and very simple to use.

TripCom has clearly shown how it responds to openness and simplicity via the general purpose 'publish and read' API and the applied data and query models that are all based on recent standards (W3C Recommendations) for RDF-based semi-structured data. The fact that computing at Web scale requires particular characteristics from a distributed middleware solutions called upon a special scalability task that started with the second year of the project. The goal of this task is to show how the TripCom models, architecture and implementation addresses and matches the scalability needs of Web service computing.

In the first part, during Year 2, concentration was put on the definition of the task, analytical evaluations in terms of distributed systems in general and the architectural approaches of TripCom. In particular, we discussed, in Deliverable D6.5v1, the scalability requirements and expectations coming from the two use cases adapted from the Enterprise Application Integration and Healthcare domains, respectively [4][8]. In Chapter 2 of this document we revisit the major results that were presented in the first version of the deliverable in order to reconsider some of the made claims, but in particular to repeat the status the scalability task had after the first year.

The objective of the second year of the scalability task is the application of the theoretical results from year one to the prototype, and to the last releases of the use case implementations. In order to facilitate the evaluation in larger settings, we prepared the necessary deployment infrastructures to run TripCom on Amazon EC2, and to automate the configuration of test scenarios. The EC2 deployment and the test environment are subject to Chapter 3. In Chapter 4 we present our generic scalability benchmarks and the results we were able to achieve in distributed settings. We did, as it will be shown in Chapter 4, not manage in all aspects to achieve the desired or expected results. In many respects this was not mainly related to scalability problems that arise by applying TripCom principles at Web scale, but rather with technical problems that resulted from the implementation. Different design choices

have unfortunately shown to be limiting the application of the TripCom infrastructure in large scale scenarios. Further details in this respect will also be discussed in Chapter 5, where the use case prototypes are analyzed with respect to the evaluation results on Amazon EC2.

The deliverable concludes with a scalability-driven analyzes of future application areas of TripCom technology. Although the TripCom prototype shows some obvious flaws, the project as a whole achieved results that will produce significant impact beyond the life-time of the project. We exemplify by means of the newly launched FP7 projects SOA4All and LarKC, and the semantic tuplespace developments at Nokia Research how TripCom impacts middleware solutions for upcoming Future Internet infrastructures and scenarios. The chapter shows that the relevance of a combined semantics, tuplespaces and Web service approach, as it was researched and fostered by TripCom, is highly relevant in various respects. Not only the specifications and architectures of TripCom are reusable, but in particular the achievements and lessons learned in terms of scalability and distribution influence further work in the domain. In this respect we recognize that TripCom and the TripCom scalability task clearly live on beyond the end if their own existence.

2 Towards a Scalable Triple Space v1 - Revisited

A first important role of the initial deliverable was to define the scalability task and to determine what scalability in the context of Triple Spaces is. The scalability task was given by means of six steps that we list, for completeness, here after again.

1. **Define overall objectives** The overall objectives of the scalability task are defined in accordance with the description of work.
2. **Specify core concepts and definitions** Describe the TripCom view of scalability in detail and sketch the general approach to evaluation.
3. **Identify scalability factors, assumptions, trade-offs** Provide a complete description of the scalability factors and the associated trade-offs.
4. **Develop and analyze configurations** Next, the factors are taken over as parameters for the development of configurations and analysis of Triple Space implementations.
5. **Realize Triple Space** Use the results of the Steps 3 and 4 to realize a scalable Triple Space.
6. **Evaluate results for scalability** Evaluate the final system with respect to its scalability, based on the given procedures and concepts.

The first five steps were extensively discussed in the first deliverable, while the sixth step is the main purpose of the second version of the scalability deliverable D6.5v2. Before turning to the contribution of this deliverable, we shortly recap the main insights and results gained during the first year that were presented in D6.5v1. Understanding these results is important for the continuation of this deliverable.

In the context of TripCom we defined scalability for Triple Space as a property which allows a triplespace system to maintain acceptable performance by expanding in a graceful way by adding components to handle increasing demand for memory, bandwidth, operations, and users. Increased frequency and parallelism of requests require a proportional increase in the bandwidth of the system once initial bandwidth limitations start degrading performance. Bandwidth may be increased by increasing the bandwidth of individual machines (up to a point), by distributing the triplespace among machines, and/or by replication of data on multiple machines. Distributing a triplespace over multiple machines requires additional internal messaging, storage, and processing. The situation becomes even more complicated, as we have particular environmental assumptions to take into account for TripCom. The Triple Space platform shall provide a communication and coordination middleware that fits the core characteristics of the Web: openness, distribution, scalability at Web dimensions, dynamism and decentralization.[1]

We therefore recognized that various scalability factors must be taken into account in order to be able to make statements about the scalability expectations of a Triple Space installation. The term scalability factors is used in TripCom to denote the functional and non-functional properties that influence the scalability of a distributed semantic space deployment. It is moreover important that these factors are considered in integration rather than isolation. In this context there is the well-known CAP theorem by Brewer that states that a system can have at most two properties out of

[1]Detailed descriptions of TripCom's interpretation of each of this traits can be found in D6.5v1 from March 2008.

Consistency, Availability or network Partition tolerance [3]. This theorem is probably the most renowned piece of work that highlights the fact that there is never just one single solution to the implementation of a distributed infrastructure. It is a well known fact that replication improves availability, and consequently it is a natural decision to distribute the work and data load across distributed servers, also in order to bring the system closer to the users [6]. However, if consistency requirements are high, then scalability and availability have to be weighted against each other [16], as they work against consistency.

A similar trade-off exists between availability and completeness, as they are understood for semantic spaces. We distinguish two distribution strategies, one being partitioning and the other one being the aforementioned replication of data. While replication leads to the issue with consistency, partitioning influences the completeness guarantee of a space. The choice of whether to rely on partitioning or replication depends both on the resources available in a given deployment (storage, CPU) and on the non-functional requirements of a given application.

Further non-functional properties such as latency, correctness or reliability and their trade-offs were discussed in the first deliverable. A last property that we would like to reconsider explicitly in the context of this scalability deliverable is security. A similar triangle as the CAP one was described by Gladman in [5], who explains that a system cannot ensure scalability, functionality and security the same time – he too proposes a two out of three approach in this respect. As the security-functionality-scalability triangle in [5] shows, we would need to weaken either the functionality of our semantic spaces, or the scalability. In many aspect the sought investigation was however about the trade-off between scalability and functionality, and aim for the former without weakening the functional properties more than what is necessary. We thus had to abstain from security in some distributed scenarios in order to concentrate on the understanding of the scalability-functionality relationship.

In order to simplify the process of addressing and eventually understanding the functional and non-functional properties of a Triple Space, we introduced the concept of configurations. A Triple Space configuration provides a means to describe the functional and non-functional properties of a triplespace realization. It thus gathers the required and, on a secondary level, the desired traits of a triplespace. The configurations are therefore a primary instrument for the analysis of Triple Space at design time. In the first deliverable we had proposed a parametric design-based analysis tool, where the various relevant properties become parameters; e.g. DistributionMechanismParam to reflect the type of distribution to apply (partitioning, replication), DistributionSizeParam that represents the number of kernels, and various non-functional indicators to determine the degree of guarantee that can be given. Examples of non-functional properties that become parameters are LatencyParam, OverheadParam to have a measure of communication overhead due to distribution or synchronization, ConsistencyParam, CompletenessParam, or AvailabilityParam. Last but not least we included two parameters to indicate the acceptable capacity in terms of triples, and the feasible load in terms of clients: CapacityParam and LoadParam. In particular the latest two are only of indicative nature, as they depend largely on the available resources of individual kernels that might not be known in detail.

Further inputs to the parametric design task are constraints and requirements. Both of these inputs are transformed to logical expressions which restrict or define the search space. One of the constraints is for example the available storage capacity

that cannot be surpassed: storage size is limited by the capacity of an individual kernel in case of replication, or by the number of kernels times the individual capacity in case of partitioning. The requirements on the other hand, also transformed into logical expressions, are directly derived from the Triple Space configurations that are provided by space developers and engineers; e.g. availability guarantee should be higher than 90%.

Concerning step 5 of the scalability task, the realization of a scalable Triple Space, there were various decisions taken towards a semi-decentralized approach of deployment. The Triple Space deployment architecture is designed so that the logical operations on Triple Spaces can be translated into interactions between network addressable physical hosts. The architecture is represented in Figure 2.1. It consists of three layers of abstractions: users, triplespaces, and physical layers.

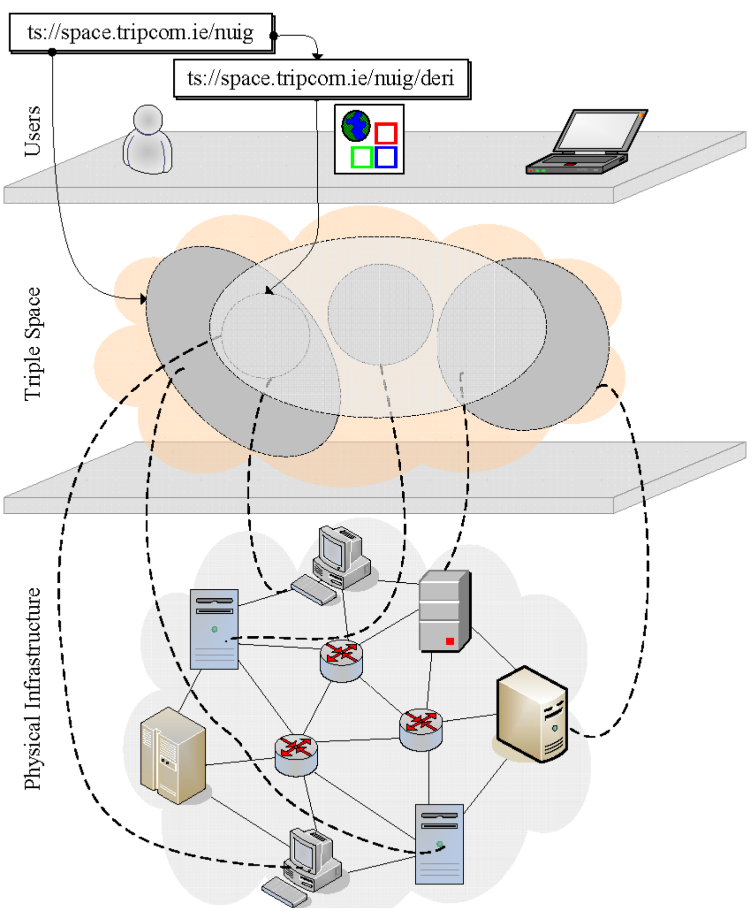


Figure 2.1: Triple Space Deployment

The user layer consists of the users and the clients of the Triple Space infrastructure. These users communicate (either locally or remotely) with the Triple Space layer through TS kernels. The Triple Space layer consists of hierarchies of triplespaces. Space hierarchies support scalability in that multiple spaces allow clients to separate communication among spaces, and coordination models can be developed which attempt to minimize the communication load on a single space by splitting data coordination into sibling spaces (where there is no overlap) or subspaces (where there is overlap). In that way, TripCom guarantees at least local scalability per interaction pattern. In the Triple Space layer, the triplespaces are managed by TS kernels identifi-

able with URLs. These URLs are resolved to the IP addresses and ports of the physical hosts in order to resolve the access points to the relevant kernel instances. Last but not least, the physical layer consists of the physical devices required for deploying TS kernels. It mainly consists of physical entities such as servers and network endpoints. The server on which a TS kernel is deployed is identified through its IP address and port number.

As this architecture allows addition of new hardware and software resources without hindering the existing configuration and with local indices built at each kernel, the overall performance can be kept at an acceptable level, and replication, system dynamism, openness, and reasoning at kernel- or triplespace-level could be supported. In addition, security measures can be enforced through the kernels.

Not only the deployment architecture was recognized to influence the scalability, but also, as shortly stated above, the functional properties. In this respect we mainly focused on the impact of the different operations that the Triple Space API offers. As a results thereof we divided the API into three levels of expressiveness that were termed Core API, Extended, respectively Further Extended API. The Core API is supposed to guarantee Web scalability in that it only addresses a priori only single kernels per operations;[2] i.e. publishing to one kernel, or reading a single triple from one kernel. The Extended API implies first restrictions, as those operations deal with multisets, and thus querying multiple kernels becomes necessary if spaces that belong to hierarchies are addressed. It is no longer enough to only return the first matching triple, but multiple matches are desired that potentially reside on multiple kernels. The same counts for the notification service that needs to be implemented on all kernels hosting a given space hierarchy. The Further Extended API is then most likely no scaling, as removal or transactions require synchronization across multiple kernels in order to ensure the desired level of consistency.

Latest for operations of the Extended API distribution and discovery mechanisms are required. In fact, as soon as there are subspaces to be queried or there are requests expressed for data without the indication of a particular target space, the Distribution Manager with support of the Metadata Manager have to intervene. P-Grid, a scalable peer-to-peer indexing systems is applied to creating routing tables for particular resources. In combination with the structural metadata about space hierarchies that is maintained by the Metadata Manager, this allows for the discovery of spaces and data in distributed settings. Furthermore, distributed spaces required the management of query distribution plans. A Query Preprocessor, as part of the Distribution Manager, acts as a query optimizer for the query decomposition and distributed query evaluation. The Query Preprocessor provides modules for query rewriting, cost estimation, join and selection ordering, query decomposition and answer construction.

Further architectural aspects that were discussed are the role of security and the storage framework with respect to scalability. The security approach of TripCom ensure scalability with respect to the execution on a single kernel. Justified concerns were however raised with respect to security and scalability in distributed settings. As the Gladman triangle shows, there is a non-avoidable trade-off between scalability and security, as the later always requires some degree of centralization. In most of the upcoming evaluation scenarios of this deliverable, we will thus not enter the subject of security.

[2]In distributed scenarios as the ones implemented by TripCom further kernels might get involved and become necessary in order to forward requests to the responsible kernel.

With respect to the storage framework, the scalability depends largely on the concrete requirements of the usage scenarios. Depending on the applied indexing, for example, the number of spaces per kernel can become a neglectable parameter. The space identifier is a field of the assumed data model and can enter the indexing process for the price of a much bigger index, and hence further storage requirements. Further fields of the extended RDF data model that is used by the OWLIM storage layer allows for integrated management of metadata. Instead of reification or increased numbers of triples as consequence of the added metadata, the applied data models integrate such statements in the triplesets. This lowers the number of triples significantly and increases the scalability of the storage layer.

The first deliverable concluded with some first statements about the expected evaluation procedures, indicators and environments that are now subject to this deliverable. The quantitative aspects of scalability were described on the basis of three dimensions, namely

- the computational resources R available,
- the load L on the system, and
- some performance measures P.

A particular aspect of scalability is captured by the observation of how P is affected when larger R have to compensate for more L. A constant remaining value of P when simultaneously increasing L and R by the same degree would express optimally the desired scalability. By defining relevant load parameters, resources parameters, and a set of performance measures, we set the ground for evaluating the performance and subsequently the scalability of the TripCom implementation. The definition of the various parameters constitutes the first step of the TripCom scalability evaluation procedure:

1. specify performance model parameters (relevant parameters to describe load, resources, performance measures),
2. review the performance model,
3. collect performance data (relationship among performance, load, resources) via experiments, and
4. analyze results.

The first two bullets were thus subject to the first deliverable, while the latter two will be presented in the continuation of this document.

3 Scalability Test Environment

In this chapter, we describe the scalability test environment and the tools employed to test the TripCom kernel.

3.1 Amazon EC2

The Amazon Elastic Compute Cloud[1] (EC2) is a Web service which provides a custom application environment on a set of distributed machines. The EC2 is a virtual computing environment in such that the number of machines, the application executed, and the network access (e.g., the firewall settings) can be configured dynamically through a Web service interface. The actual reservation and allocation of a physical machine is transparent to the user.

The user's application environment within the computing cloud is represented as an *Amazon Machine Image* (AMI). An AMI is a virtual machine image and therefore contains the operating system, all applications, libraries, data and configuration settings. It acts as a "template" for a virtual machine (machine instance) to be created and started. In order to allow for the dynamic loading of AMIs via the Web service interface, they have to be uploaded to Amazon's *Simple Storage Service* (S3). Multiple AMIs with different configurations and applications can be uploaded to the S3. The user can start, monitor, and terminate as much instances of these interfaces as needed.

As a prerequisite of using the Amazon Elastic Compute Cloud, an institution must hold a (free) amazon.com account. By default, each account is limited to a maximum of 20 concurrent instances of an AMI which is too few for our requirements.

3.1.1 EC2 I/O and CPU Performance Guarantees

AMIs run on so called instances, virtual machines that run on physical machines in the Amazon datacenters. Users can choose between different instance types to fullfil the resource and compute power needs of their AMIs. For the TripCom kernel, we chose the "Small Instance", which has the following properties: 1.7 GB memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB instance storage (150 GB plus 10 GB root partition), 32-bit platform, I/O Performance: Moderate, Price: $0.10 per instance hour.

A "compute unit" is described as the "equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. This is also the equivalent to an early-2006 1.7 GHz Xeon processor referenced in our original documentation. Over time, we may add or substitute measures that go into the definition of an EC2 Compute Unit, if we find metrics that will give you a clearer picture of compute capacity."

Typically, a physical machine in the amazon data center provides more than one instance, thus, running instances share physical resources such as network and the disk subsystem. Amazon states that "if each instance on a physical host tries to use as much of one of these shared resources as possible, each will receive an equal share of that resource. However, when a resource is under-utilized you will often be able to consume a higher share of that resource while it is available".

[1] `aws.amazon.com/ec2`

As a consequence, actual network and disk performance may vary between instances. Moreover, shared resources are not evenly shared. An I/O performance indicator ("moderate" or "high") is given with each instance type. Instance types with "high" I/O performance have a larger allocation of shared resources than the ones with "moderate".

3.1.2 High-Level Architecture

Triple Space kernel deployment on Amazon EC2 is a multi-step process, explained in detail in the following list:

(1) Create AMI As mentioned before, an AMI is the virtual machine image that is used to allocate machines and create virtual machine instances. We therefore created a new AMI (Triple Space AMI: `ami-2bb65342`), installed necessary prerequisites for kernel execution such as a Java virtual machine and an SSH server for remote interaction, and stored the resulting image into Amazon S3. Note that this step has only to be performed once, all following steps are executed multiple times, i.e. once per test or even multiple times during testing in order to increase the number of machines/kernels during test runs.

(2) Allocate Machines A machine on EC2 is allocated using the following Web service API call: `ec2-run-instances ami-2bb65342 -k gsg-keypair`, using our AMI ID and a keypair for remote interaction via SSH. The result of this call is an *instance id*, identifying the machine instance that was created as a result of this call. This instance however is not instantly available, it has to be ensured through polling via `ec2-describe-instances i-ae0bf0c7`, that the machine is booted and has received a valid public DNS name. Once the machine is booted and ready, the Amazon firewall has to be configured in order to allow incoming connections from outside the Amazon network (e.g. through `ec2-authorize default -p 22`).

(3) Install/Start Kernel As a next step, the Triple Space kernel is copied over, installed and started on the newly allocated and booted machines via a series of `scp` and `ssh` commands.

(4) Validate Kernel State Since kernels build a large distributed system, we have to ensure that each kernel that is started is actually part of that system, ensuring that it is correctly connected to the P2P network and can be discovered and reached by other kernels. This requires a script that is part of the kernel itself and validates its state, e.g. checks the PGrid routing table, reachability of other nodes, etc. This script is then invoked after starting a kernel; if it returns an error, the kernel is reported as failed so that we can manually connect to the instance and fix the problem.

(5) Run Clients Once at this point, test clients can be started that create and populate spaces with test data and conduct test runs in order to measure performance and scalability characteristics. See Section 3.2.1 for details on the test clients that perform these steps. Note that Step 2 and 3 of this section may be performed multiple times during test runs as the number of active kernels/instances needs to be increased during each cycle.

(6) Shutdown Instances As currently an instance-hour for the smallest instance type on EC2 costs USD 0.10, all instances need to be shut down and removed from the EC2; this again is automated using the EC2 Web service API (`ec2-terminate-instances i-ae0bf0c7`).

3.1.3 Custom Tooling

For the implementation of these steps, we use a combination of multiple layers of shell scrips (higher-level scripts use commands implemented by lower-level scripts) and custom command line programs that interact directly with the EC2 Web service API exemplified by the Web service API calls in the text above. Using this approach we can automate the full kernel deployment process from machine allocation and kernel installation to validation in one single shell script on the highest level.

3.2 Test Environment

3.2.1 Scalability Clients

This section describes the application which has been developed to measure the scalability of the TripCom prototype (the application is called "scalability client"). The scalability clients use a triplespace to receive their configuration and store the results of the measurement. They are generic and not restricted to measure the outcome of a specific operation. Instead, the configuration of the client contains, besides other options, the operation which shall be measured (see below for further details). This configuration is represented by RDF triples which is communicated to the clients via a triplespace. When a client receives its configuration, it executes the benchmark according to the configuration and collects results (i.e. the runtime of the operation). After a successful execution it writes the measured data into a triplespace. Additionally, the client writes a triple into the configuration space indicating that it has finished. This triple can be used to synchronize the execution of multiple clients.

Currently, a client can be used to measure out, in and rd operations. Depending on the measured operation different configuration parameters are required. The following list shows which parameters have to be provided:
The following configuration parameters are used independently of the operation which shall be measured.

requestType the type of operation which shall be executed by the client (i.e. rd, out, in).

rootTestSpace the space which shall be used to execute the operation.

startingTime (optional) the time when the operation shall be started. If the time is in the past or the value is not set the client starts immediately.

endingTime (optional) the time when the operations shall end. If the value is smaller then startingTime or in the past, the value is ignored.

repetitions (optional) the number of repetitions. Either repetitions or endingTime has to be set. If both values are set the client issues an error.

dependsOn (optional) this parameter can be used to indicate that the client has to wait for another client to finish before it can start its measurements. If this value is set, the client waits for the client on which it depends to write the triple which indicates that it has finished.

The following parameters are only valid if an out operation is measured.

usesTriple (optional) the triples which shall be outed by the client. The triples have to be provided in RDF/XML format.

usesFilename (optional) the name of a file which contains the triples (in RDF/XML format) to out. Either usesFilename or usesTriple has to be set.

The following parameters are only valid if an rd or an in operation is measured.

query the query which is used to retrieve data from the space.

Execution of Benchmarks

This section describes step by step the execution of scalability benchmarks using the scalability clients.

First, clients connect to their configuration space and issue a blocking in operation to wait for their configuration. The URI of the configuration space has to be provided as parameter during the startup. This first step of the client execution is illustrated in figure 3.2.1. We decided to use blocking in operations instead of notification because it is possible that the configuration is written to the configuration space before the clients are started. Using notifications a client would not get its configuration if the triples are already in the space when its started. After starting the client it waits for its configuration as long as nobody shuts it down or writes the configuration into the space.

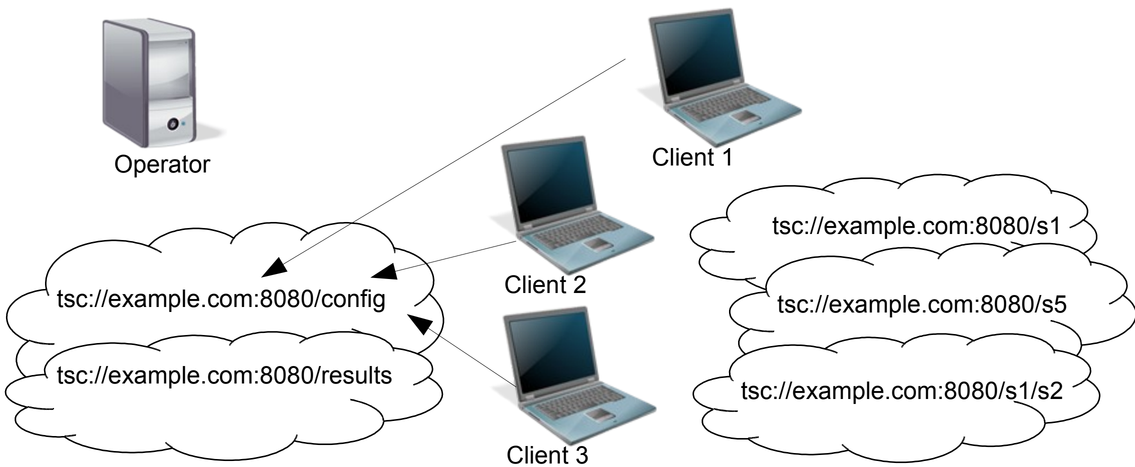


Figure 3.1: The scalability clients issue a blocking in operation to be informed when a new benchmark shall be executed (i.e. a new configuration for the client is written to the configuration space.

The second step of the client execution is that some application (called “Operator” in the figures) writes the configuration for the clients. This step is shown in figure 3.2. The client tries to the retrieve its requestType argument first. After it retrieved the type of operation which shall be measured it tries to take the other configuration arguments according to the request type (see section 3.2.1 for the list of parameters). Listing 3.1 shows an example configuration of a client which measures an out operation. The clients name is “Client2” and it writes triples from a file into the space “tsc://localhost:8080/test2”.

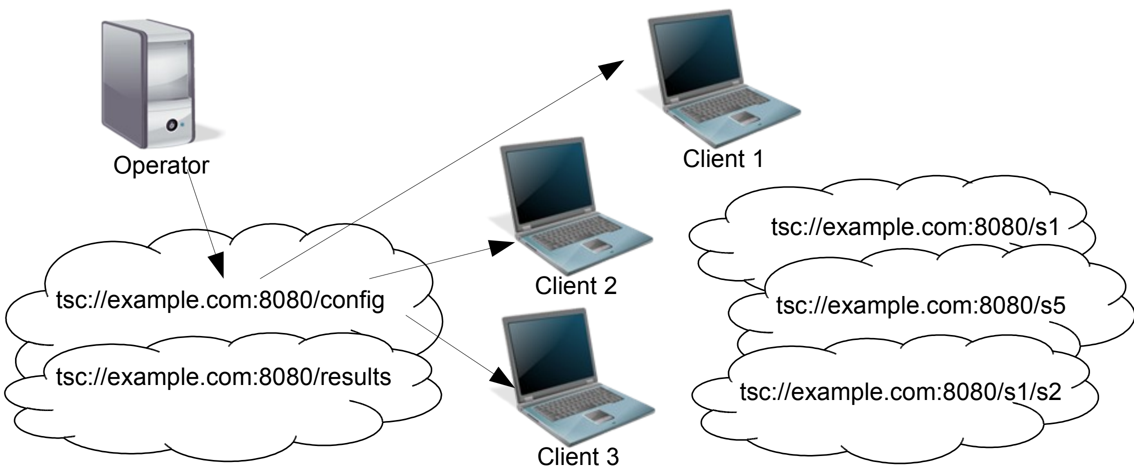


Figure 3.2: The operator writes a new benchmark configuration into the configuration space and the clients receive the configuration.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3          xmlns:scal="http://www.tripcom.org/scalability/">
4      <rdf:Description rdf:about="http://www.tripcom.org/scalability/clients/Client2">
5          <scal:requestType>OUT</scal:requestType>
6          <usesFilename xmlns="http://www.tripcom.org/scalability/">
7                  testdata/VideoAssetsFiles/100VideoAssets.rdf
8          </usesFilename>
9          <scal:rootTestSpace
10             rdf:resource="tsc://localhost:8080/test2" />
11         <scal:startingTime>1221128057880</scal:startingTime>
12         <scal:endingTime>1221128117880</scal:endingTime>
13     </rdf:Description>
14 </rdf:RDF>
```

Listing 3.1: Client configuration of a client which measures an out operation.

After the clients received their configuration they execute the benchmarks. During the execution they measure the execution time. This step is illustrated in figure 3.3. If an error occurs during the execution the client will not write data into the result space. The error is only logged at the machine which executes the client and the “operator” machine is not informed about the error.

The last step in the execution of the clients is that they write the measured data into the result space. An interested party (in the figure it is the same machine which wrote the configuration) can read the data from the space and use it for further processing. This final step is depicted in figure 3.4. In addition to the measured data, the client writes a triple to the configuration space which indicates that the client has finished with the measurement. This triple can be used by other clients to synchronize their

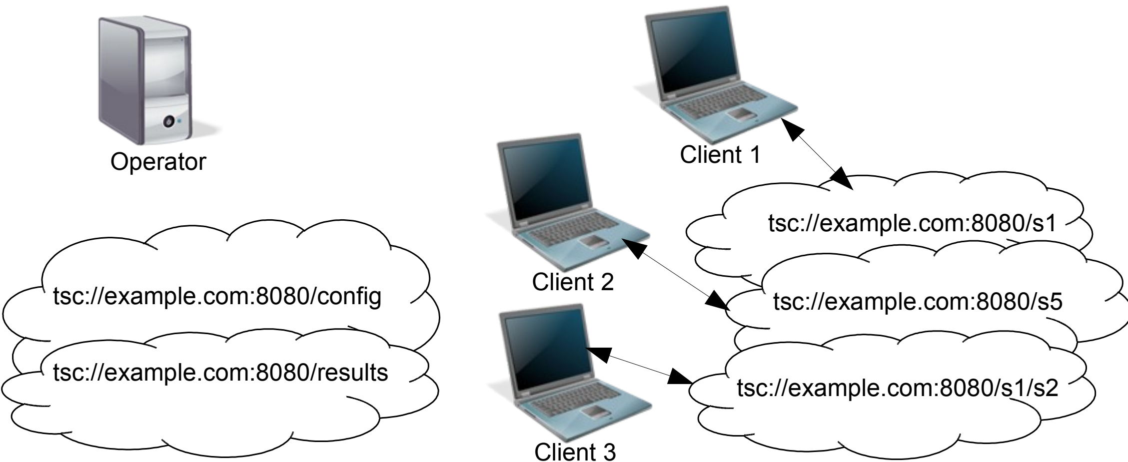


Figure 3.3: The scalability clients execute the benchmark on the spaces according to the configuration and measure the results.

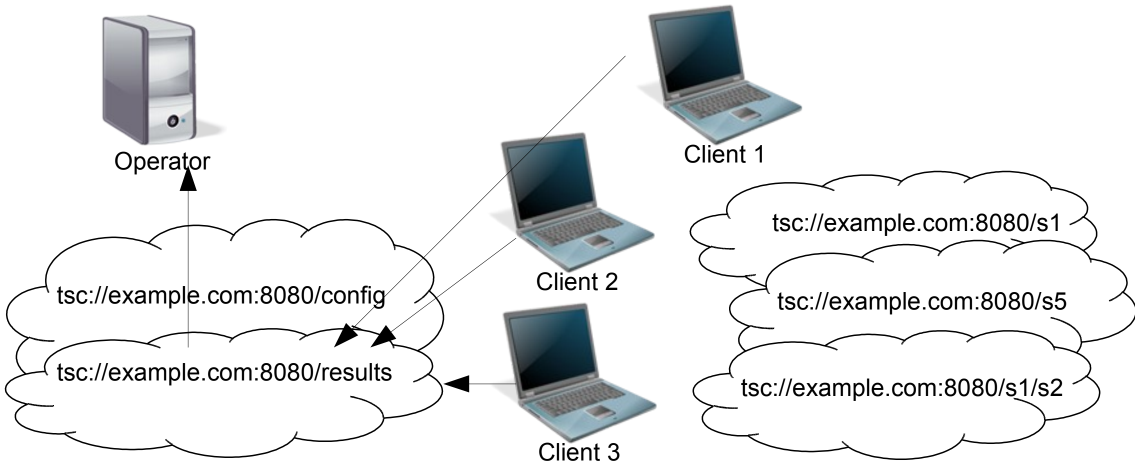


Figure 3.4: The clients write the measured values into the result space and the operator collects the results for further processing.

execution. For example consider a client which outs data into a space and another client which measures the time it takes to rd this data from the space. It is clear, that the outing client has to finish the out operation before the reading client can start with its benchmark. If the reading client would start to early, the rd operation would block and this would tamper the measured results. In order to solve this problem the "dependsOn" configuration parameter has been introduced. After the reading client retrieved its configuration it waits for the triple which indicates that the outing client has finished. Not till then the client starts with its execution. Therefore, it is assured that the outing client is executed before the reading client.

4 Evaluation of the Prototype

This chapter describes generic scalability measurements which have been performed to evaluate the design and concepts of the TripCom prototype.

4.1 Generic Scalability Benchmarks

This section describes several benchmarks which have been performed to evaluate the scalability of the prototype. These benchmarks do not evaluate the kernel with respect to a certain application or use case. Instead, generic benchmarks are used which show the scalability of the prototype design and the concept of subspaces.

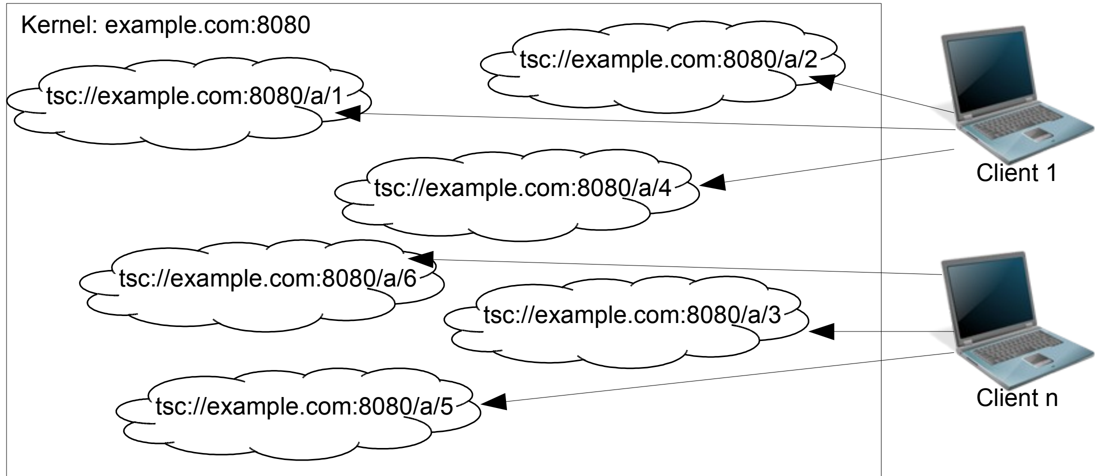


Figure 4.1: One kernel hosts several spaces which are used by multiple clients. Even when the client only uses a separate space it is influenced by other clients because all spaces share the resources of the same kernel.

The TripCom prototype supports spaces which can have multiple subspaces. A subspace does not have to be located at the same physical machine as its root space. Instead, it can be located on any (reachable) kernel instance. By distributing the subspaces, the scalability of an application can be improved dramatically. Additionally, the distribution of spaces can happen completely transparent to the clients. Clients only use the space URI to address spaces. Usually, a client is connected to any kernel in the system. If a client tries to use a space which is not physically located at the kernel to which it is connected the kernel issues a lookup operation to find out on which kernel the space is located. Afterwards, the kernel transparently forwards the request to the correct kernel. This concept is illustrated in Figures 4.1 and 4.2. The first figure shows a kernel which hosts 6 spaces (the rootspace of the spaces is not shown). Two clients concurrently use these spaces. Using this setup all 8 spaces share the resources (processing time, memory, . . .) of the kernel on which they are hosted. Therefore, an operation on one subspace influences an operation on another subspace even when the applications, executed by the clients, do not have any relationship. The second figure shows the same setup, but the spaces are distributed over two kernel instances. Using this setup, operations executed on subspaces located at one of the kernels do not influence operation on subspaces on the other kernel. As shown below, the scalability

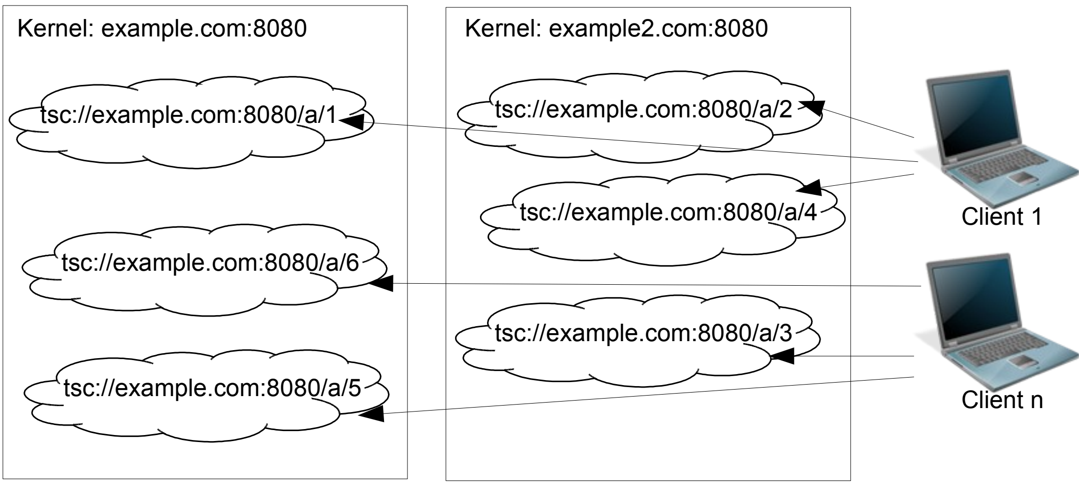


Figure 4.2: Two kernels host several spaces which are used by multiple kernels. In contrast to the setup shown in figure 4.1 a better performance can be achieved because the kernels can work in parallel.

of an application can be improved by locating the subspaces on different machines and making sure that the application connects to the kernel on which the subspace is physically located.

4.1.1 Read Operation in Various Distributed Settings

This section presents the results of the generic scalability benchmarks described in Section 4.1.

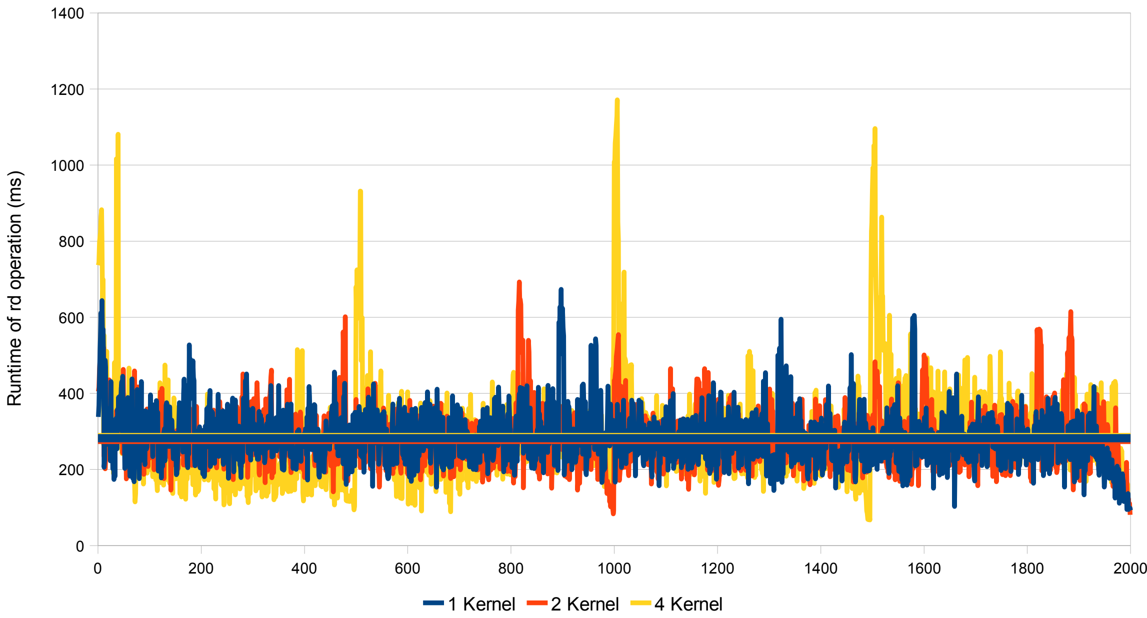
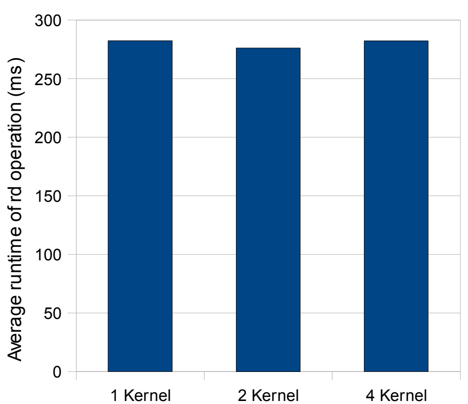


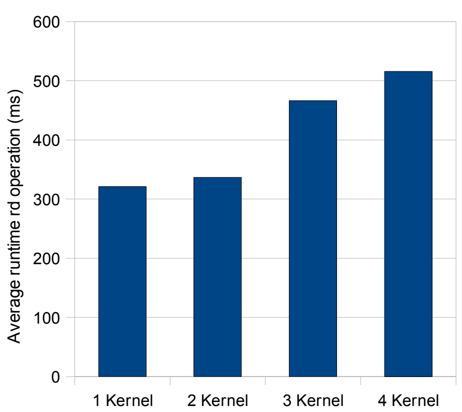
Figure 4.3: Measurement of read operations using one to four kernels.

In order to equalize the partly very much varying execution times of the operations, all measurements were done based on 2000 executions for each of the setups. Figure 4.3

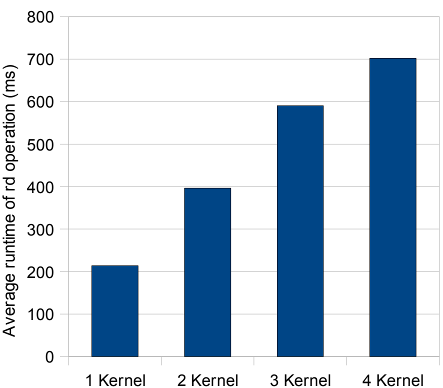
shows all 2000 measured values for one of the read evaluations. In this benchmark, ten subspaces are equally distributed over one, two and four kernels respectively. Since the experiment is supposed to measure the scalability and not primarily the performance of the infrastructure, the test scenario increases the number of clients equally to the number of kernels. Initially, there were ten concurrent clients that accessed one kernel which hosted 10 spaces. In the second setting, twenty clients and two hosting kernels are used, while the third one applies a configuration with forty clients that access spaces on four kernels. As it can be seen in Figure 4.3, the measured results vary quite impressively. This exemplifies very nicely the consequences of exploiting a tuplespace infrastructure as integration middleware within a kernel (cf. [9]). The tuplespace does not guarantee that concurrent operations are executed in the order in which they have been issued. This means that it is possible that operations are overtaken by other operations, which can delay the processing of individual calls significantly.



(a) The clients are connected to the kernel which hosts the used subspace.



(b) The client is connected to a randomly selected kernel. The subspace might not be located on this kernel.



(c) All clients are connected to the same kernel which has to route the requests to the kernel which hosts the subspaces.

Figure 4.4: Illustration of the generic scalability measurements for read on distributed kernels.

Figure 4.4 illustrates the results of the scalability measurements. As mentioned above, the indicated responsiveness measures depict the average performance of 2000 measurements. The x-axis of the graphs show the number of kernels in use for the

measurements and the y-axis the response time of the operation in milliseconds. The first graph (Figure 4.4(a)) shows the results for a benchmark on one, two and four kernels, which is as such comparable to the test environment configurations that led to the results shown in Figure 4.3, with slightly different numbers of clients. The number of clients that execute the 'rd' operations increase at the same rate as the kernels in the distributed deployment. There are eight clients reading from one kernel, 16 clients that read from two kernels and finally 32 clients that access four different kernels on four EC2 instances. This scenario showcases the constant performance in terms of latency, as the work load on individual kernels remains constant. In other words, also the overall work load and traffic in the network increases linearly, the TripCom infrastructure easily compensates by increasing the number of kernels that handle the load, as long as the clients that execute the read operations are directly connected to the kernel on which the targeted spaces are hosted. As a consequence there of, it is not necessary to forward any requests to other kernels.

The second graph (Figure 4.4(b)) summarizes the results of a configuration that has the clients interact randomly with one of the available kernels. Consequently, there is only a $\frac{1}{\#kernels}$ probability that the space would be hosted locally to the contacted kernel. As the figure shows, the response time increases when more kernels are in the network, as the probability that the request has to be forwarded to another kernel increases. As this particular evaluation setting not only increases the number of kernels, and hence the number of forwarding traffic, but also the number of clients that in addition increase the work load, the overall perceived latency increases faster than proportionally to the number of remote retrieval. In fact, the processing latency starts to dominate the communication delays.

Finally, Figure 4.4(c) shows the result of the measurements when clients are connected to one and the same kernel. This kernel has to process all requests from all the clients and forward them to the appropriate kernels on which the targeted space are hosted. Again, the probability that an operation can be processed locally is $\frac{1}{\#kernels}$. As shown in the figure, the response time of the operations increases more rapidly with the amount of kernels and clients compared to the previous case because the host to which the clients are connected becomes a bottleneck.

4.1.2 Read Operation Under Varying Workload

This section describes the results of scalability measurements with respect to the number of clients, and thus with varying numbers of requests. In contrast to the previous sections, the amount of kernels is fixed for each of the measurements and only the number of clients increases. The benchmarks have been executed with one, four and 16 kernels, each hosting one top level space (so-called rootspace) and 8, 16, 32, . . . , 512 clients respectively that query the distributed infrastructure. As for all experiments, each kernel runs on its own virtual EC2 machine. The collected data thus indicates the scalability of the Triple Space kernels with respect to the number of concurrent clients.

Figure 4.5 shows the results of the measurements with one kernel and the increasing numbers of clients. Since the number of clients augments exponentially, the y-axis is given in logarithmic scale. The achieved measurements in terms of response time give proof to a linear behavior with respect to the increase of concurrent clients. This linear scalability in terms of infrastructure and work load is required for TripCom to be

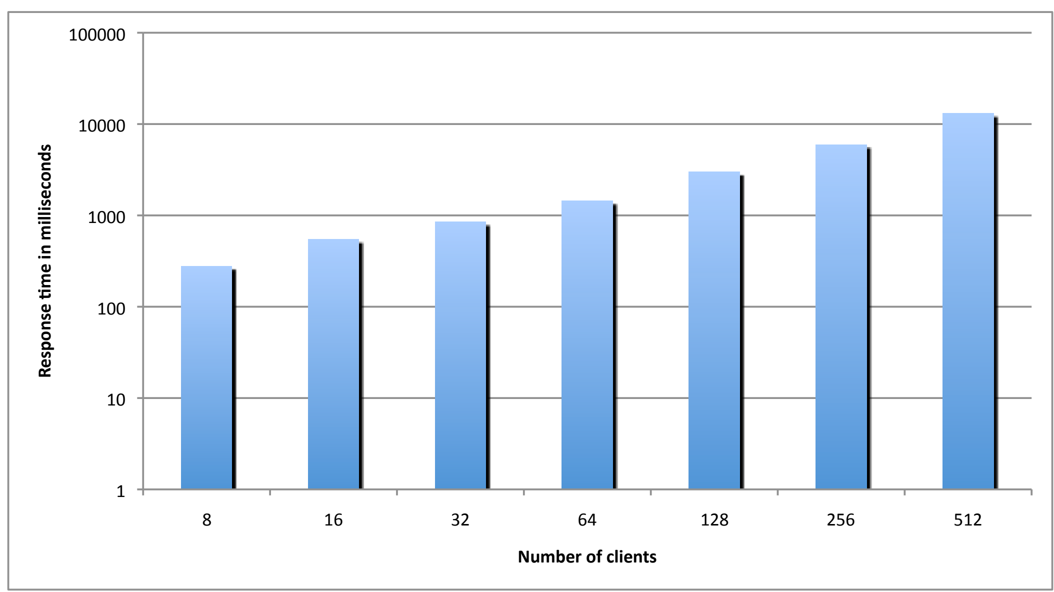


Figure 4.5: Latency measurement of read operations on one kernel.

applicable in Web-scale environments, and we conclude that the infrastructure scales well with respect to the number of concurrent clients. Note that the measurement only reflect the load in terms of requests, i.e. client interactions, and not in terms of data load. For evaluations in this respect, the reader is referred to the respective deliverable of WP1 that evaluated the performance and scalability of the persistency infrastructure [15]. In fact, in the context of retrieval, the data load only influences the transmission time (which is not in the hands of TripCom), and the data management at the level of storage.

Table 4.1: Read from one, four and 16 kernels and up to 512 clients.

	1 kernel	4 kernel	16 kernel
8	277,67	216,01	211,56
16	549,21	424,48	437,81
32	849,26	819,79	889,92
64	1451,39	1454,58	1394,46
128	2994,73	2607,92	2502,87
256	5942,56	4739,53	4825,50
512	13183,53	9021,18	9030,07

The results of all benchmarks are presented in Table 4.1. First of all, the results show, as already discussed above (Figure 4.5) that the response time of the read operation increases roughly linearly with the number of clients. Comparing the latency in terms of increasing number of kernels shows that for up to 64 clients the measured delays are nearly equal and thus independent of the number of kernels. In the same range of tests, the results also indicate that the increase in latency is slightly sub-linear. The reason for this observation is simple. The kernels' capacities are not used fully, and the behavior is comparable to the one observed in the tests of the previous section. However, when 128 concurrent clients access one kernel, our test machine comes closer to its limits. Consequently, the overall increase in response time of the

operation starts to grow observably faster in a setting with one kernel in comparison to the settings that rely on more than one kernel. The advantage of multiple kernels begins to unfold its capacity because the load is distributed over the available kernels. For example, the tests with 256 clients show an average delay of around 13 seconds on one kernel, why the answers arrives already 4 seconds faster in a network with four kernels. Obviously, requests cannot be handled four times faster in systems with four kernels, as the distribution of load and spaces requires additional communication overhead which adds the previously observed forwarding delays. Furthermore, it can be observed that while the latency deviates between the one-kernel and the four-kernel, respectively 16-kernel scenarios, the latter two show the same measurements for up to 256 clients. This is again explained with the saturation barrier that seems to lay beyond 256 clients in a 4 kernel deployment. Figure 4.6 summarizes again the just discussed result of the measurements with one, four and 16 kernels. Note that this figure does not have any logarithmic scale, but it nicely depicts how the latency increases much faster in the non-distributed deployment. In other words, the TripCom realization with respect to distributed spaces on distributed kernels seems to fulfil its goals.

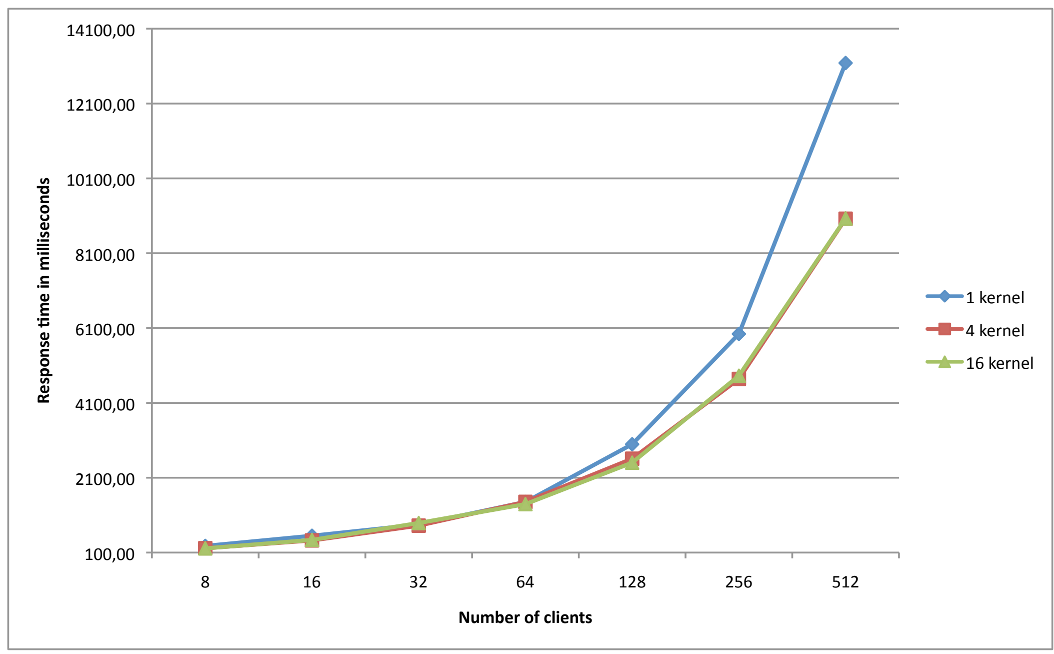


Figure 4.6: Summary of read under increasing work load.

4.1.3 Read Operation Without Space Indication

The following evaluation results depict the retrieval operation in the context of a non-targeted 'rd' call, i.e. a read without indication of the triplespace to read from. In addition to what was showcased in the previous sections, this time the Distribution Manager of the kernel needs to apply the collected indexing information to retrieve the requested data. Figure 4.7 shows latency measurements for a simple case for which there is no remote communication necessary. In other words, the information to be retrieved is stored locally, i.e. the index points to kernel that is already involved in the retrieval operation. It can be easily seen that four the one, two and four kernel settings,

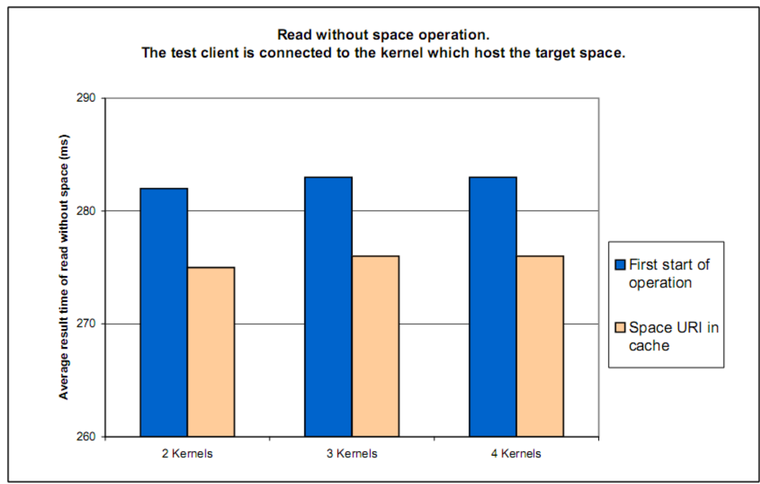


Figure 4.7: Read without target space indication - local discovery.

the overhead for indexing is almost neglectable and that the results are comparable to the previous sections. A further minimal improvement can be gained, already in these rather small-sized scenarios, by applying the added caching mechanisms (cf. [10]).

Although, the indexing overhead shows already quasi-constant scalability in the case of local retrieval, the advantage of caching becomes more evident in distributed scenarios. As for the previous sections, the scenario of Figure 4.8 involves remote fetching of information; i.e. the targeted kernel first has to query the index to retrieve the identifier of the space to query, and then has to forward the request to the kernel that actually hosts the desired data. Due to the remote querying, this scenario inherits the linear increase in latency that was already observed previously. Again, in the case of one, two and four kernels, respectively, the indexing overhead is quasi-constant. This is easily explained by the fact that P-Grid has proven to be in the logarithmic complexity class with respect to search across nodes. In other words, the indexing mechanisms that are implemented in the Distribution Manager add a logarithmic component to the otherwise linear retrieval complexity, which remains the dominant aspect. Finally, taking a look at the smaller bars, we can observe the claimed improvement by means of caching.

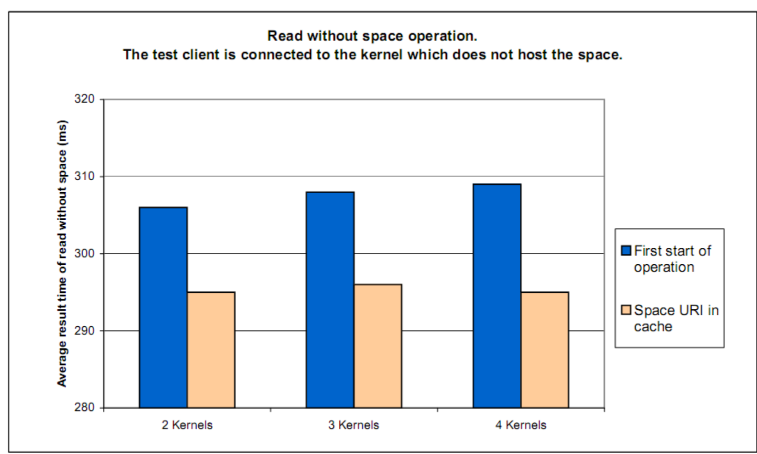


Figure 4.8: Read without target space indication - remote discovery.

4.1.4 Publication via 'out' Operation

The simple 'out' operation, as it is defined in the Core API showed in principle the same performance figures as the simple 'rd' operation. In fact, publication in that simple sense is simply the inverse operation of retrieval. While for the showcased read, the request must be forwarded to the right kernel in the distributed Triple Space system, for the publication operations it is the triple that must pass through the system. This requires thus the same kernel discovery procedures and communication overhead. Eventually at the target kernel, 'out' triggers a store operation on the repository, while 'rd' invokes the query engine. In terms of the overall latency, these two operations are comparable, as again, it is the communication latency of the network, i.e. the network latency, that is the dominating factor. We thus conclude, that also 'out' has proven to be at least in the linear complexity class.

4.1.5 Multiset Operations of the Extended API

The TripCom Extended API contains operations over multisets, such as reading multiple triples, or 'rdmultiple'. It also includes retrieval operations with more sophisticated templates, such as full SPARQL queries in the context of our implementation, and the publish/subscribe support.

In this subsection we shortly discuss the expected and observed behaviors of the TripCom prototype in this respect.

Multiset out: The 'out' operation with a set of triples to publish turned out to have the same performance and scalability as the simple out operation of the Core API, up to a given number of triples. In fact, 'out' triggers the forwarding of data to the storage framework of the kernel that hosts a space. The only difference between the core 'out' operation and the one that supports sets of triples is the size of the messages, and internally to a kernel, the size of the JavaSpace entries. The scalability is thus not limited by the distributed Triple Space, but rather by the scalability of a single kernel instance in terms of the number of simultaneously written triples.

Rdmultiple/Rd with SPARQL: In the simplest case both of these operations behave as the 'rd' primitive of the Core API. With the simplest case we refer to the setting where a request is resolved by a single kernel instance, as all searched data is stored on the same machine. Requests in TripCom are resolved by the query engine that is shipped with the OWLIM repository, and there it does not make any difference if a triple pattern is resolved or a full-fledge SPARQL query. Also the former must be transformed to a SPARQL query in order to be adequately treated. A similar argument accounts for 'rdmultiple' in the non-distributed context. The storage framework can return one results ('rd') or multiple results ('rdmultiple') with the same procedures, and thus with the same effort. A decrease in scalability is however observed for both operations, once the infrastructure has to operate over distributed spaces, i.e. over multiple subspaces on distributed kernels. The difficulty lies in the fact that the distributedness of the subspaces is a priori not known. A subspace on a different kernel can have further subspaces on further distinct kernels. The TripCom architecture does not give any limits to this structuring. The decrease in scalability and thus the reason for having these operations in the Extended API are the following:

- SPARQL queries must be processed (Query Pre-Processor) and distributed across multiple kernels, and eventually the results must be merged to reflect a common virtualized view across all addressed spaces and kernels.
- Subspaces are parts of their parent space, and any subspace must a priori be queried upon a 'rdmultiple' request to a space with children. As stated above, these subspaces can potentially be distributed across multiple kernels with an unknown depth of subspaces. Depending on the complexity of the hierarchy, this adds significant communication overhead, and notable effort in terms of subspace discovery. While this operations is thus quite indeterministic in its behavior, TripCom defines a non-recursive variation of 'rdmultiple' that does not take into account the space hierarchies. As explained above, this non-recursive variation has the same performance and scalability as the core 'rd' primitive.

Publish/subscribe: The publish/subscribe support in TripCom is implemented as part of the storage framework. In other words, the storage framework, through the TS Adapter, takes care of storing subscription and of informing clients about subsequent matches. As long as a space is hosted by a single kernel, the overhead for this procedure is very small and a kernel internal issue that does not impact the overall scalability. However, once a space is distributed, all kernels of all distributed subspaces must jointly ensure the handling of subscriptions. This again, similarly to the situations above, increases the expected scalability, and publish/subscribe ended up in the Extended API.

4.1.6 The Further Extended API

The Further Extended API was defined to group the Triple Space operations with the least scalability expectations, however, with the most complex functionality. In other words, the Further Extended API is the direct counter part to the Core API that was tested and evaluated in the beginning of this chapter. The arguments for this claim are simple: removal (given by the 'in' operation and the support for transactions require synchronization across multiple kernels in order to ensure the desired level of consistency. Synchronization in turn increases the communication overhead and computational complexity of the Triple Space system, which hampers performance and eventually the scalability.

5 Evaluation of the Use Cases

In the following sections, we apply the results and discussion of the prototype evaluation to the two use case scenarios, and discuss the delivered performance and scalability measures in the context of the use cases.

5.1 EAI

The EAI use case has focused in the implementation of a marketplace model using Triple Spaces in the context of the TripCom WP8A. The concrete example used (a digital content marketplace), could be easily extrapolated to any other service or product offering following the same marketplace business pattern. In the WP8A the need for a middleware with a minimum functionality to build this marketplace has been motivated, and a result, our implementation has used the Extended API configuration of the Triple Space kernel [9]. As a result, we inherited the trade offs motivated in the previous version of this deliverable related to less scalability in exchange for the functionality needed (more query expressivity concretely).

5.1.1 Marketplace Design Concerns

During the implementation of our marketplace prototype, we expressed some concerns related to design decisions which affect the scalability and the marketplace exploitation model. They are analyzed here, since they have served as a basis to identify two potential deployment models, as we will see later in this section.

Security Constraints A real marketplace implementation has very severe constraints about some data accessibility. Some information (e.g. client information, billing information) is obviously neither modifiable nor visible by other parties. There is a second security level for information which might be public but no modifiable by competitors or final customers in order to avoid malicious use of this data. We must take into account that there is a clear trade off between security and latency. The more security checking needed, the more time it will be needed to answer a query. Furthermore, these heavy security restrictions might derive in a federated deployment of the marketplace, as we will comment later. With respect to the former security restriction, one might ask if the information is confidential, why it should be published in the middleware. Assuming that the marketplace implementation aims at coordinating requests and offers from many requesters/providers, only information needed to fulfill this purpose should be taken into account by the system. With respect to the latter, it seems clear that we will have to cope with the scalability trade offs imposed by this need of security policies.

Uncertainty of Potential Users One major constraints of the system is that it is very difficult to predict how many customers will make use of the marketplace, making client scalability is a crucial issue for a marketplace deployment. Fortunately, the read evaluation results (see Section 4.1.2) show almost a linear response time with concurrent client accesses given that they connect to the kernel which has the information (the performance is worse if the system has

to forward the request to a different kernel to which the client is sending the query to). This leaves the scalability in terms of client growth as a matter of an appropriate space hierarchy definition, which can be handled by the marketplace owner, as we will see later.

Completeness of the Results Retrieving all the potential offers might be a critical issue in terms of marketplace usefulness. As argued in the first version of this deliverable, there is a clear tradeoff between completeness and availability / latency. Completeness is heavily penalized when the spaces are distributed in several kernels, and therefore a space specialization can benefit this issue. This, however, limits the scope of offers to the size of information that can be handled by a single kernel, which will restrict the marketplace size, but that is a decision the marketplace owner would have to face.

As a result of these analysis, we can present two contrary deployment models we will compare next.

5.1.2 Marketplace Deployment Models

Given that we plan to implement a commercial marketplace model with Triple Spaces, we have several providers publishing offers that will be consulted by clients to match their requests. The information storing in a marketplace context can be separated in information that can be stored via a bath process (like the content and service catalogue), and real time information (such as contact modification or auctions). The amount of information that needs to be processed in real time is lower, and can minimize the impact of the out response time described in 4.1.4.

There is a decision to make in the sense of moving the information to a federated deployment (see Figure 5.1), in which each provider provides information in its own kernel, which is shared to the marketplace. This deployment would be narrow the influence of the marketplace owner as a mere entry point for client request. Only the information shared by all the providers (i.e. film information in our case) is really authorized by the marketplace owner. This deployment represents a chance for a better control of each provider data, since it is directly authorized by each provider and stored in their own kernels. However, this deployment provides two clear trade offs: the space structure is left to each provider, and thus unknown by the clients. This imposes the use of many read without URL operations, which will heavily penalize 1) latency, and 2) completeness, which were clear requirements of the marketplace (see the impact of this operation stated in Section 4.1.3. As we see, however, caching mechanisms can palliate the effect of distributing the query and finding out the kernel which has the information we are querying.

On the other hand, a centralized deployment (see Figure 5.2, assumes offer information is authorized by the marketplace owner, which implies the space hierarchy definition is also made by the marketplace owner, and the information can be stored in more rational way (see how the clients can directly access to spaces depending on the kind of information they need to know). This solves the concerns presented before, and as the scalability results show, can provide a good scalability in terms of client growth and concurrency. Completeness also benefits from this approach, since the information is not distributed and is thus easier to retrieve. In this case the read operation used would be a read multiple, which has the implications discussed in Section 4.1.5.

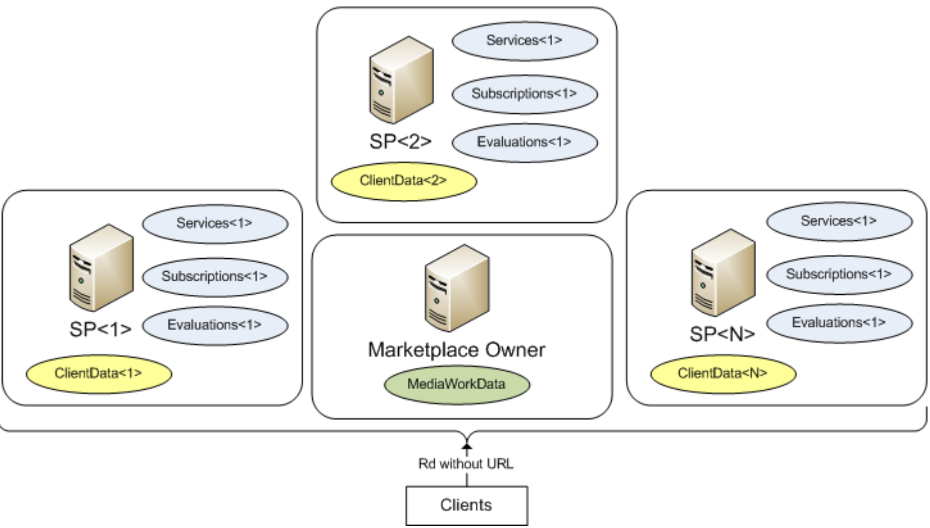


Figure 5.1: Federated Deployment of a Marketplace

There is, however, a major tradeoff. Having the information specialized in one kernel, imposes two major restrictions: 1) the amount of information that can be stored, and 2) load balancing is penalized, since the requests are centralized. The latter restriction could be treated by the replication of kernels, although it would also imply consistency trade offs. The former restriction is inherent to the deployment model chosen

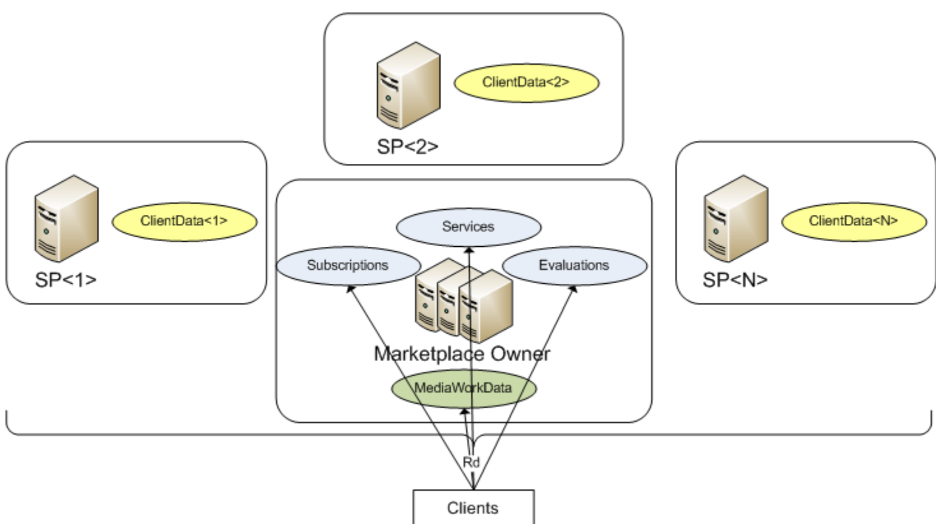


Figure 5.2: Centralized Deployment of a Marketplace

5.1.3 Recommendations

As a conclusion of this analysis, we can extract some recommendations. The impact on the completeness and latency of the first deployment model makes this solution not very appropriate for a marketplace implementation. Specializing spaces, and defining a space hierarchy similar to the presented in the second model seems a better choice, but provide a clear conclusion for us. The size of the marketplace information must be limited if implemented with Triple Spaces. Otherwise, the completeness and latency requirements imposed won't be met and scalability will be useless.

The other side of the coin is, therefore, that a marketplace implementation restricted to our functional constraints (the use of the Extended API configuration), with the aforementioned security, uncertainty of number of users, and completeness of results constraints, can scale only to a certain point. Widest marketplaces (such as eBay) would have to be divided into different independent sub-marketplaces, in order to cope with a Triple Space based implementation. The reason, is, as we have seen, inherently related to the trade offs scalability imposes on these factors: security, latency and completeness.

5.2 eHealth

The EPS scenario can be implemented with different Triple Space configurations, and each design choice about the structure of the spaces has an impact on scalability. In D8B.2 [1] and D8B.3 [2] two different ways of achieving the goal of an European patient summary are described. The requirements for a fully functional and integrated EPS infrastructure, dealing with the existing state of the art of large-scale healthcare systems, are:

- data ownership: the configuration must allow participants to manage data according to their own laws and organizational processes.
- security: the configuration must allow participants to be compliant with their laws about privacy and security of medical data.
- integration with existing systems: the EPS infrastructure should be considered as an integration platform rather than a new infrastructure posing new requirements to all the countries participating the EPS.

The two different scenarios that we are going to illustrate are a meaningful example of the trade-offs between scalability and availability that have been explained in the introduction. The first one shows that scalability can be achieved by relaxing the requirements of data distribution and security, while the second one shows that scalability can be affected when completely satisfying the domain requirements.

5.2.1 Patient Summary on a Single Kernel

The first scenario, described in D8B.2, is based on the assumption that patient summary data belongs to the patient himself. A patient, in turn, is in charge of a single health authority, which is responsible for managing his medical data. This means that, regarding data ownership, all the participants to the infrastructure agree that while clinical data has to be stored according to national laws about privacy, summary data can be sent to the authority that is in charge of a patient, even if it's in a different country with respect to the originating one. Such a scenario could be reached only with a strong political agreement among the countries involved in an EPS initiative. This choice is expected to offer higher scalability, by diminishing the complexity of the configuration, at the expenses of the data ownership and security requirements, which are relaxed.

Under these assumptions, there can be two possible spaces hierarchies, depicted in figures 5.3 and 5.4. In both of them, data of a patient summary is always stored on

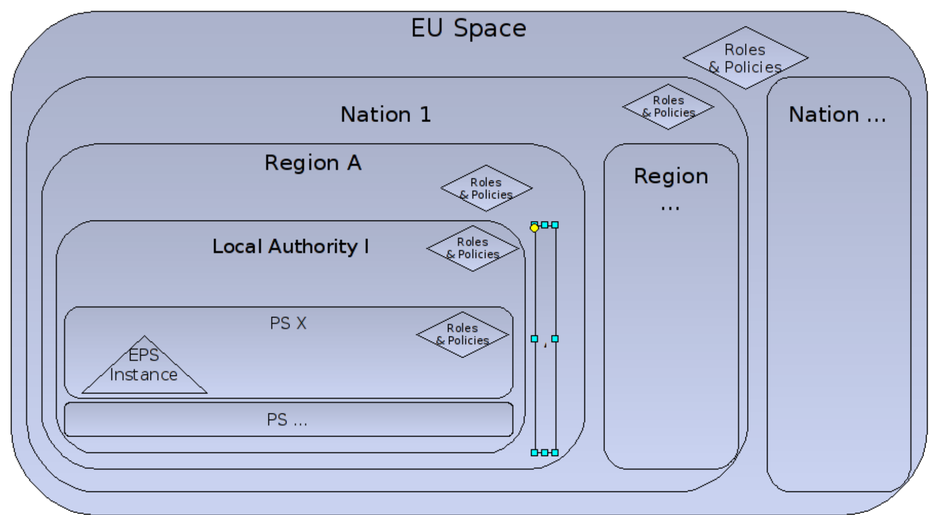


Figure 5.3: Patient-based scenario with nested security policies

a single kernel, and is always in the leaf spaces of the tree. The only difference lies in the implementation of the security policies. In the first case, each health authority has its own space with its related security policy. When accessing data, all the security policies from the root space down to the desired space are evaluated, thus enforcing global and local policies. In the latter case, each kernel has to directly and coherently implement all the policies starting from the national ones.

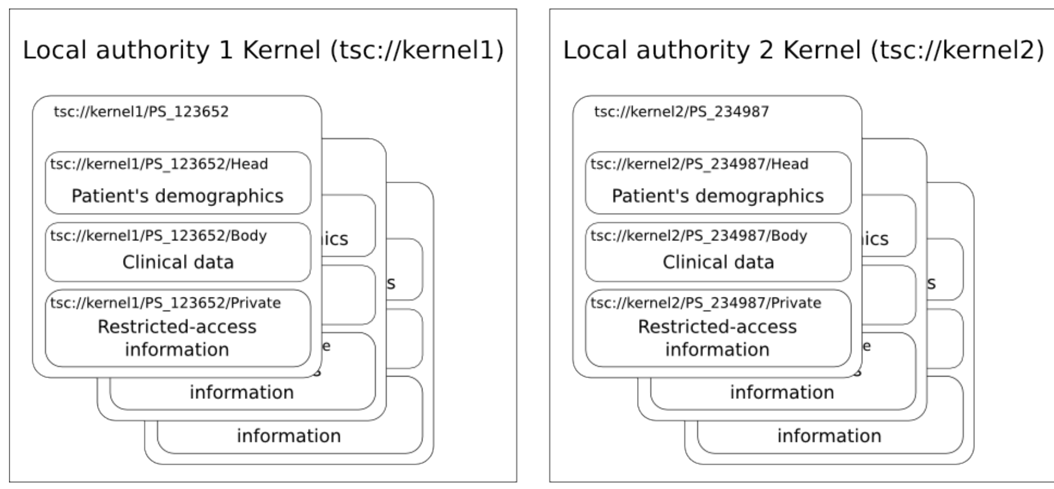


Figure 5.4: Patient-based scenario with independent kernels

Together with the usage of the "location triples" described in D8B.2, this configuration allows locating the space hosting a single patient summary with a single Core API (read without space) call and retrieving PS data with an Extended API call (SPARQL query to a space hosted on a single kernel). The SPARQL query will be targeted to the main PS space, and so it will always retrieve data from just four spaces (the main one, plus its descendants). When increasing both the number of kernels and of requests, a read operation without a target spaces, according to the results presented in 4.1.3, scales logarithmically. A normal read with a target space hosted on a single kernel, as explained in 4.1.1 and 4.1.2, instead scales linearly.

When using a "single tree" configuration, policy evaluation depends on the level of the healthcare organization responsible for the desired patient summary (the lower the organization is in the hierarchy, the greater the number of policies evaluation will be). Anyway, the depth of the tree in this configuration will remain stable in time, thus allowing to exploit the caching mechanisms of the Security Manager component. As higher level policies will be always evaluated, they will likely be cached, resulting in a lower response time. The results of the scalability evaluation carried out for the Security Manager component, explained in D5.4, show a linear decrease of throughput when increasing the number of levels in the hierarchy of spaces.

Following the load indicators provided in the evaluation plan (cf. [13]), a single kernel will sustain a traffic of 10.000 requests per day. With such a load, given that a patient summary is hosted on a single kernel, we can conclude that with this configuration the system can meet our requirements in terms of performances.

5.2.2 Patient Summary on Multiple Kernels

The second scenario, described in D8B.3, deals with an European patient summary infrastructure composed of existing national healthcare systems, each one implementing a minimum set of changes required to integrate with the Triple Space without changes to the security model or the data distribution strategies.

Under these assumptions, the structure of a patient summary becomes quite complex, as also depicted in figure 5.5 (each of the represented spaces can be hosted on a different kernel). Each of the three main spaces of a PS contains several subspaces (to a maximum number equal to the number of countries in the EU), and each of these subspaces has an arbitrary structure, as it is located on a remote kernel completely managed by the remote healthcare authority. The number of spaces contained in a single patient summary becomes not predictable at design time, because each country can have a different, and eventually complex, data distribution policy that is reflected in the spaces structure.

In the example of figure 5.5 we see an example of such a complexity. It depicts the hierarchy of the spaces containing data of an Italian patient living in the United Kingdom. This particular patient has also lived in Lombardy (Italy) and in the Netherlands, so there will be demographics data hosted on English and Dutch kernels. There are also clinical data originating from each of the three countries, whereas private data is only from the United Kingdom and the Netherlands. In this case, the spaces will be hosted on these kernels:

- Kernel of the Lombard healthcare authority
- Kernel of the Lombard hospital 1
- Kernel interacting with the Dutch Hub
- Kernel hosting data from the Dutch GP
- Kernel of Dutch clinical center 2
- Kernel interacting with the English NHS

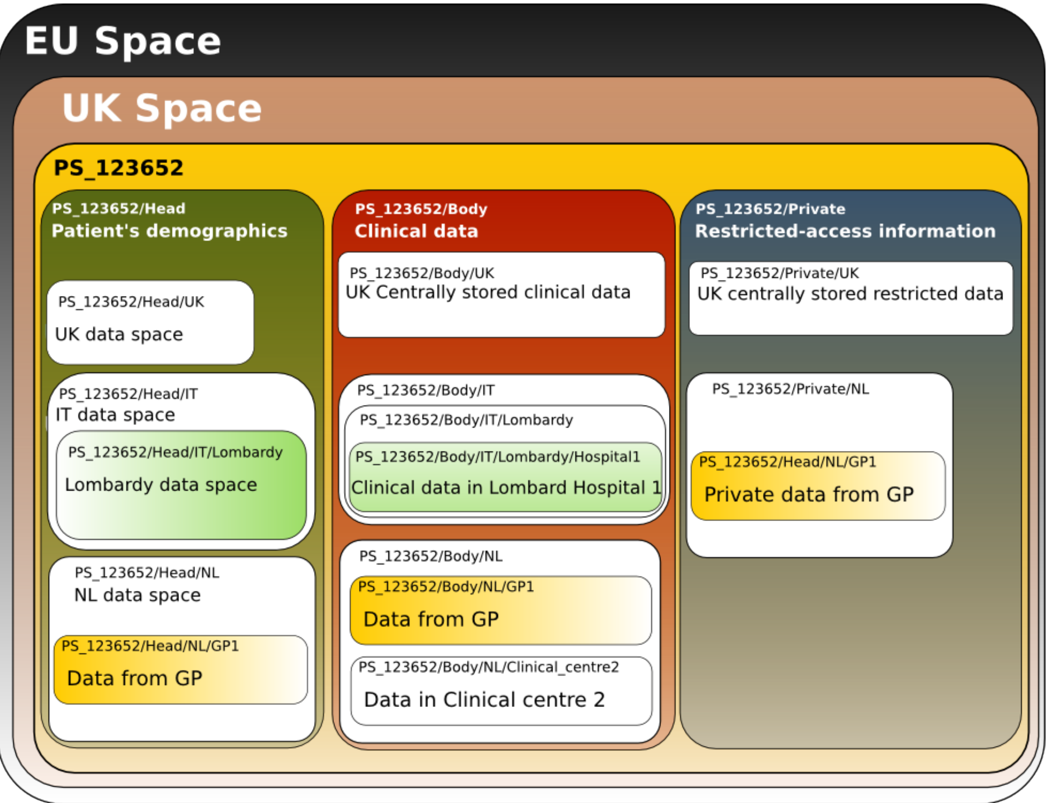


Figure 5.5: Spaces structure when integrating existing systems with minimal agreement on a reference structure of the spaces

The spaces are distributed over 6 kernels. This means that when querying, the query will be forwarded at least 6 times, thus resulting in a performance loss of a factor of least 6 and a corresponding increase of the load on the kernels participating to the EPS. Writing new data inside an existing patient summary structure instead is not affected by the issues of the read operation, because a write operation has always a precise target space and is not recursive, so we can apply the results described in 4.1.4, concluding that even in this worst-case scenario we can meet the requirements for writing new data inside a patient summary.

With this configuration, we meet all the requirements of the EPS in terms of data ownership and security enforcement in a worst-case scenario when the integration with real-world systems happens without modifying any architectural decision of these systems. Anyway, the complexity of this particular configuration influences the performances, but this is because the scenario considers the integration of existing systems, respecting the peculiarities and design decisions of each system without taking any step in the direction of a consolidation or a simplification.

6 Scalability Issues Beyond TripCom

The TripCom project achieved significant results in bringing tuplespace computing in combination with semantic technologies to Web scale. In this way the project delivers an important contribution to the state-of-the-art of semantic computing in the large, in particular in the context of service computing with Semantic Web services. Such achievements paved the way for recently launched projects and endeavors.

The so-called Future Internet pushes technology forward in three distinct, but eventually blurring domains: i) the family of Web services and Semantic Web services constitute the Internet of Services, in which software and applications become ubiquitous; ii) the Internet of Things, in which in principle every physical object becomes an addressable resource on the Web, and iii) the Mobile Internet that grants seamless connectivity over multiple devices on an anywhere, anytime basis. In fact, pushing the Web towards mobile and in the wider scope ubiquitous computing (also referred to as the Ubiquitous Web [14]) is seen as a core pillar of upcoming Web realizations: multiple devices jointly enable a user's Web experience by reusing and composing available data and services as desired. All of these upcoming flavors of the Web require TripCom-style technology to cope with the increasing complexity and size of the Web in terms of data and users. The conversion of semantics and the Web-style 'publish and read' communication paradigm are seen to be TripCom's main enabler of scalability in this respect. In fact, semantic technologies are the key enablers for at least partial automation of many of the central task related to the management of data, users and services; an indispensable process for keeping the Web simple, open and scalable, in spite of the envisaged evolution towards the Future Internet.

In the continuation of this chapter, we discuss scalability-related issues in efforts that are conceptually related to TripCom, but reach beyond the scope of this project. In particular we address work in the area of mobile computing, large scale reasoning and Web-scale service-oriented architectures. One common denominator is the trend towards virtualization and the blurring of infrastructural services such as communication and storage in cloud platforms.

The TripCom project influenced the Semantic Web-related research of the Nokia Research Center in Helsinki. Sedvice, one of their recent developments is a combination of space-based computing and the Semantic Web [11, 12]. The core idea of the Sedvice system is to establish the Semantic Web vision in form of shared, but localized and potentially personalized spaces. This approach is supported by the argument that the Semantic Web is characterized by dynamicity and monotonicity of information, and the fact that the meaning of knowledge is depending upon the locality, the use and the user. A space is then the virtual closed world in the scope of which information is perceived according to a common context. The global Semantic Web is seen as being the union of the many small and localized information spaces. Sedvice provides the architecture and a lightweight implementation for the realization of these concepts in mobile and ubiquitous environments. The primary goal is to enable control-flow free interaction mechanisms for distributed agents running on mobile devices.

Although Sedvice targets the Semantic Web, its scalability requirements, in terms of individual spaces, are clearly smaller than the ones of TripCom. Nokia expects their spaces to handle thousands of triples that are relevant to a given user in its specific context. In TripCom, the expected load on single spaces, or space hierarchies is ex-

pected to exceed the millions, or even billions of triples. Still, in particular because Sedvice is installed in the context of the mobile or ubiquitous Internet, it clearly shows that many of the TripCom concepts and results point towards the future of the Internet; both in terms of functionality, as much as scale.

The aim of the EU FP7 Large-Scale Integrating Project LarKC is to develop the Large Knowledge Collider, a platform for massive distributed incomplete reasoning that will remove the scalability barriers of currently existing reasoning systems for the Semantic Web. The platform will run on a computing cluster and will also consider alternative computing paradigms such as "computing-at-home". It will be realized through a pluggable architecture that gives users the possibility to develop plug-ins in order to test heuristics and techniques from diverse areas such as Web search, Semantic Web, databases, machine learning and cognitive science, among others. The LarKC interfaces define five major plug-in types:

- Identify: this plug-in identifies the data sets that could be used in order to answer a users query.
- Transform: this plug-in is capable of transforming some piece of data from one representation to another.
- Select: this plug-in is responsible for narrowing the search space even further than Identify has done by taking a selection (or a sample) of the data set that has been made available by Identify on which reasoning should be performed.
- Reason: this plug-in executes a given SPARQL query against a set of triples provided by a Select plug-in.
- Decide: this plug-in is responsible for building and maintaining the pipeline containing the other plug-in types, and managing the control flow between the plug-ins.

According to the initial operational framework of LarKC, the project consider the plug-ins to share the large volumes of data involved in the reasoning process by 'passing' the data sets from one to another via a shared data layer [17]. In other words, one plug-in passes a reference to a set of triples to the plug-in that is responsible for the next processing step. The LarKC platform thus requires from its data layer to offer efficient storage and access to data and the related meta-data (in sum some billions of triples), and the core functionalities non-surprisingly match largely the key characteristics of TripCom:

1. persistence and retrieval of data through a simple and standardized API, optimized to deal with huge volume of information (e.g., streaming support);
2. import from (parsing) and export to (serialization) standards formats (e.g. all the popular RDF syntaxes);
3. standard query mechanism and infrastructure (e.g., SPARQL endpoint);
4. sharing reasoning-related metadata (e.g. attaching weights, timestamps or other system information to RDF statements).

5. possibility to support light-weight reasoning.

A shared/remote repository can reduce communication and computation costs as many reasoning tasks can be reduced to query evaluation. Communication is optimized because the volume of the query results is usually much lower than the volume of data involved in (transferred for) the evaluation. Moreover, a repository component can be distributed or parallelized under a variety of different schemata, while this remains transparent to the platform and the plug-ins. This way the data layer allows for "separation of concerns" (e.g., the storage master takes care of storage optimization, decides on distribution strategy). The first prototype of LarKC assumes a data layer that is based on ORDI,[1] which in turn is implemented on top of OWLIM [7] – a further synergy between the two projects at hand.

Although, LarKC suggested at the time of release of the initial framework to only rely on a semantic repository rather than a space-based infrastructure, the features that are requested clearly point towards a future that is more related to Triple Space than pure database technology. There is a clear coordination aspect observable, as various plug-ins have to collaborate to achieve the desired reasoning goals (massively distributed and incomplete) in the large. In terms of scalability, the future of Triple Space thus points towards the delivery of a storage and data access layer that can process several billions of triples in near real-time.[2]

In the EU FP7 project SOA4All aims at realizing a world where billions of parties are exposing and consuming services via the integration of five complementary technical advances into a coherent and domain independent service delivery platform: SOA, semantics, Web services, Web2.0 and context-awareness. While, today, service-oriented environments mainly exist within large enterprises, SOA4All seeks to exploit service-orientation on the Web in order to make the Service Web as accessible and ubiquitous as today's information Web. Service technologies have great potential to be used over the public Web to foster interoperability and innovation. In particular, SOA4All expects millions of users with a wide range of technical abilities to profit from the project outcomes; thus the "4All". The diversity of users is in various respects challenging: i) SOA4All must support and satisfy user, appliers and developers of service systems alike, and ii) it must be ready to cope with this enormous number of users and services. At the one extreme, there are those who today can only read Web pages at thus should on the Service Web be able to invoke services; and at the other extreme, those who today create a mash-up should on the Service Web be able to build a composite service. The key issue is thus to support the broad set of user roles in a manner similar to the way today's Web supports Web site creation, publication and usage.

In SOA4All a triplespace-motivated platform is considered as a core infrastructural service that is integrated with a distributed open-source enterprise service bus. The resulting infrastructure is referred to as SOA4All Distributed Service Bus (DSB). The primary goal of the DSB is to ensure that the SOA4All framework scales to the aforementioned dimensions in terms of millions of users and billions of services, by enabling appropriate distribution techniques that evolve the traditional ESB techniques towards a fully Distributed Service Bus, without altering the communication and interaction

[1] `http://www.ontotext.com/ordi/`

[2] See the LarKC technical annex, available at `http://www.larkc.eu`.

patterns of the ESB core. While the state-of-the-art ESB technology relies mostly on client-server communication, the DSB provides SOA4All services with more powerful communication patterns (e.g., publish-subscribe, event-driven, semantic-oriented) that allow further decoupling of the communicating entities in terms of time, processing flow and data schema. In particular, the integrated support for semantics thanks to the Semantic Space infrastructure empowers direct links to reasoning or mediation techniques. By adding semantic spaces, SOA4All moreover realizes an integrated infrastructure that grants access to distributed semantic repositories for service descriptions, a storage infrastructure for sharing monitoring data and other collaborative tasks, and asynchronous and/or event-driven communication without requiring SOA4All platform services and business services to take care of additional access points - i.e. the SOA4All DSB is conceptualized to provide the core infrastructural services of SOA4All in a all-in-one solution. This is an important principle for the scalable SOA4All Runtime, as it allows any type of communication efficiently and transparently over the Web by means of sharing or exchanging any type of data in between any type and number of distributed actors. In summary, the DSB exposes the traditional enterprise service bus functionality extended with scalable Web-style publishing and reading, integrated support for semantics and event-based communication mechanisms as basis for shared data management and collaborative activities.

In this way, the SOA4All Distributed Service Bus establishes a platform that could also be referred to as storage and communication cloud for large scale service-oriented architectures. In a world of more and more distributed and heterogeneous prosumers, i.e. entities that are neither traditional information or service providers, nor consumers, but rather Web2.0-style contributors, the loosely-coupled integration and virtualization of information, services and resources becomes crucial for scalable development and deployment of service-oriented infrastructures. Triple Spaces, as developed in TripCom, or the service bus of SOA4All, as presented above, are initial proposals and important steps towards that vision.

The trend in service computing goes indeed beyond the recent trend of "Software as a Service" and enters the era of "Everything as a Service" (XaaS) by binding humans, objects, and resources such as storage or computing devices as services to the global infrastructure. Information and computation are thus delivered by services and disappear behind service interfaces. The services themselves moreover disappear in the Web that becomes the platform, as open and distributed alternative to legacy systems. As a consequence of this, services become utilities, and their functionality and the offered quality of service are the relevant characteristics, rather than the endpoint. Through the cloud, functionality is now provided by the community, and no longer by individual and dedicated providers. In such scenarios, average users can become prosumers of the cloud, and the number of providing, respectively consuming entities reaches the dimensions of the World Wide Web, also in the scope of service-oriented infrastructures. In order to manage this wealth of resources, there is a significant need for metadata, and as seen from many recent software and service projects in particular also semantic metadata about service capabilities and interfaces, objects, users, and their preferences, goals or desires. The amount of semantic data that will have to be processed will likely exceed the numbers that were considered in TripCom, however, the work done in the project delivered significant insights, technologies and solutions to manage semantic data in a large-scale virtualization platform for future service infrastructures.

7 Conclusions

With this deliverable, we conclude the TripCom scalability task. The task was initially defined as reaction to the feedback received during the first project review and the projects claim to seek a Web-scale knowledge coordination infrastructure. A first deliverable was presented after month M24 of TripCom that concentrated on some of the main aspects on the levels of software design, architecture and implementation of the Triple Space infrastructure. The M24-version thus took care of the initial five of the six steps that were defined for the scalability task: definition of the overall objectives, specification of the core concepts and definitions regarding scalability and its specific meaning to the project, architecture and implementation details that allow TripCom to develop a scalable infrastructure, and indicators and approaches to the evaluation of the prototype.

The sixth and last step of the scalability task, the evaluation of the prototype, was now subject to this second version of 'D6.5 Towards a Scalable Triple Space'. In order to do so, we provided the necessary implementations to deploy the TripCom infrastructure on Amazon EC2, and defined a testing environment and test clients. A presentation of this effort was given in the first part of the deliverable. Successful results, critical observations, and problems with the implementation that could be detected in the evaluation process were presented in this deliverable. The evaluation results were applied in the context of the two use cases of TripCom: i) Enterprise Application Integration and ii) eHealth. This assessment showed that the showcase domains were well chosen and that TripCom technology brings clear advantages to the application scenarios. Still, we also had to recognize that many of the technological problems that we encountered do not allow for direct exploitation of the project results. Further effort in terms of performance improvement are thus necessary before the results can be exploited in real business scenarios.

The results that were achieved still show that TripCom is able to deliver the required linear scalability in terms of workload (doubling the amount of resources, doubles the amount of work done), and sub-linear scalability, preferably logarithmic scalability, in terms of data load. The latter is inherited from the logarithmic scalability of the indexing algorithms of P-Grid for example. Again, although we must recognize the problems with the overall performance, we managed to show the necessary scalability in order for TripCom to serves as a Web-scale infrastructure.

Finally, the deliverable made an outlook into the future, and discussed how TripCom technology is or could be applied to address the scalability requirements of ongoing and future projects that work on knowledge intensive infrastructures for the Web of Services, or the Future Internet. Most prominently, TripCom is an important precursor of the semantic cloud principle that enables the virtualization of large-scale knowledge management infrastructures on the Web. This emphasizes again that TripCom does not really come to an end, but that its results, and its approaches towards a Web-scale knowledge infrastructure are the beginning of many new and innovative efforts.

References

[1] A. Carenini and D. Cerizza and R. Krummenacher and D. Foxvog and J. Martinez and V. Momtchev and N. Pérez Crespo. Prototype of TripCom application for sharing health data among healthcare organizations. TripCom Project Deliverable D8b.2, 2008.

[2] A. Carenini and R. Krummenacher and D. de Francisco Marcos and M. Lafite and V. Momtchev. Assessment of the developed solution with regards to the detected indicators. TripCom Project Deliverable D8b.3, 2009.

[3] E.A. Brewer. Towards robust distributed systems. In *19th Ann. ACM Symposium on Principles of Distributed Computing*, page 7, July 2000.

[4] D. De Francisco Marcos, G. Toro Del Valle, and L.J.B Nixon. Towards a Multimedia Content Marketplace Implementation based on Triplespaces. In *7th Int'l Semantic Web Conference*, pages 875–888, 2008.

[5] B. Gladman. Risks to Safety and Security. Presentation at Scrambling for Safety 8, August 2006.

[6] X. Jiang, F. Safaei, and P. Boustead. Latency and Scalability: A Survey of Issues and Techniques for Supporting Networked Games. In *13th IEEE Int'l Conference on Networks*, pages 150–155, November 2005.

[7] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM - a Pragmatic Semantic Repository for OWL. In *Workshop on Scalable Semantic Web Knowledge Base Systems (WISE)*, pages 182–192, October 2005.

[8] R. Krummenacher, E. Simperl, D. Cerizza, E. Della Valle, L.J.B. Nixon, and D. Foxvog. Enabling the European Patient Summary through Triplespaces. *Computer Methods and Programs in Biomedicine*, 2009.

[9] L.J.B. Nixon, D. Martin, D. Wutke, M. Murth, E. Simperl, R. Krummenacher, B. Sapkota, Z. Zhou, H. Moritsch, C. Schreiber, O. Shafiq, G. Toro del Valle, D. Cerri, and V. Momtchev. Platform API Specification for Interaction Between All Components. TripCom Deliverable D6.3, March 2008.

[10] L.J.B. Nixon, K. Teymourian, R. Krummenacher, H. Moritsch, V. Momtchev, A. Ghioni, and A. Schütz. Semantic Clustering and Self-Organization in Triple Space. TripCom Project Deliverable D2.4, September 2008.

[11] I. Oliver and J. Honkola. Personal Semantic Web Through A Space Based Computing Environment. Invited Talk: 1st Workshop on Middleware for the Semantic Web (ICSC), August 2008.

[12] I. Oliver, J. Honkola, and J. Ziegler. Dynamic, Localised Space Based Semantic Webs. In *IADIS Int'l WWW/Internet Conference*, pages 426–431, October 2008.

[13] R. Krummenacher and E. Simperl and d. foxvog and V. Momtchev and D. Cerizza and L. Nixon and D. Cerri and B. Sapkota and K. Teymourian and Ph. Obermeier and D. Martin and H. Moritsch and O. Shafiq and D. de Francisco. Towards a Scalable Triple Space. TripCom Project Deliverable D6.5v1, 2008.

[14] D. Raggett. The Ubiquitous Web. W3C Presentation, http://www.w3.org/2005/Talks/0621-dsr-ubiweb/, June 2005.

[15] B. Sapkota, V. Momtchev, and O. Shafiq. High-Performance Storage Implementation. TripCom Project Deliverable D1.3, March 2008.

[16] A. Vijay Srinivas and D. Janakiram. A Model for Characterizing the Scalability of Distributed Systems. *ACM SIGOPS Operating System Review*, 39(3):65–71, July 2005.

[17] G. Tagni, A. ten Teije, F. van Harmelen, B. Bishop, M. Kerrigan, G. Gallizo, A. Tenschert, L. Bradesko, V. Momtchev, and A. Kiryakov. Initial Operational Framework. LarKC Project Deliverable D1.2.1, October 2008.