TAPAS

IST-2001-34069

Trusted and QoS-Aware Provision of Application Services

Deliverable D15

TAPAS QoS-aware Platform: technology and demonstration

Report Version: Deliverable D15 1.0

Report Delivery Date: September 2004

Classification: Public Circulation

Contract Start Date: 1 April 2002 **Duration:** 36m

Project Co-ordinator: Newcastle University

Partners: Adesso, Dortmund – Germany; University College London – UK; University of Bologna – Italy; University of Cambridge – UK

Project funded by the European Community under the "Information Society Technology" Programme (1998-2002)

TAPAS QoS-aware Platform: technology and demonstrations

Werner Beckmann

Adesso AG

Santosh Shrivastava

School of Computing Science

University of Newcastle upon Tyne

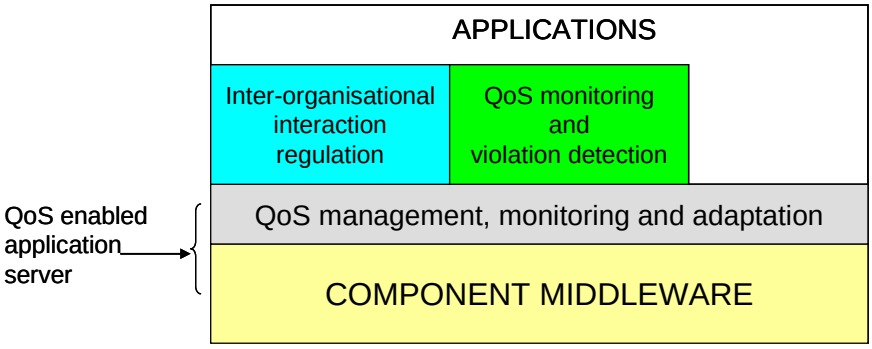
(Editors)

Table of Contents

1. Introduction

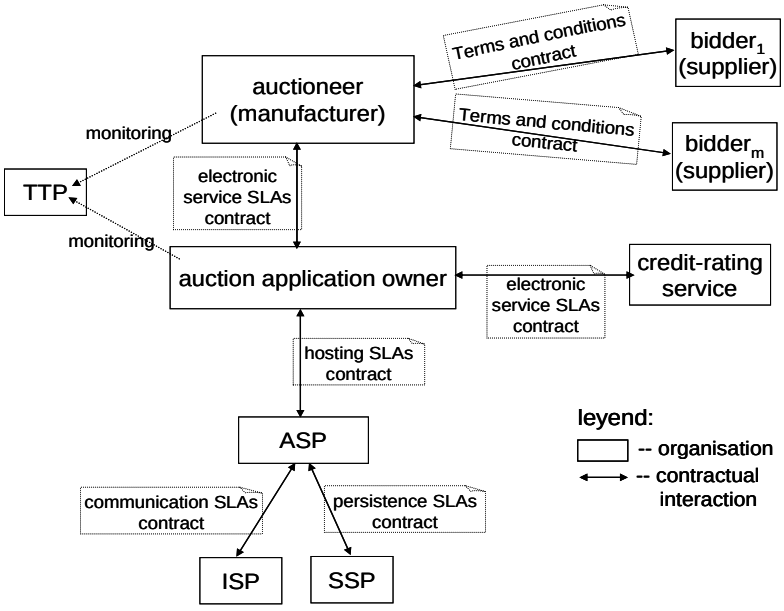
This report, TAPAS deliverable D15, describes the integration of TAPAS subsystems developed in the second year into a working platform, and demonstrator applications built for testing and evaluation.

The figure shows the main features of the TAPAS architecture. If we ignore the three shaded/patterned entities (these are TAPAS specific components), then we have a fairly 'standard' application hosting environment: an application server constructed using component middleware (e.g., J2EE application server). It is the inclusion of the shaded/patterned entities that makes all the difference.



TAPAS Architecture

A distributed auction application has been implemented to illustrate various features of the TAPAS architecture. Chapter 2 presents the overview of the application and various contracts.



Auction Application with participants and their contractual relationships

The auction application runs on a clustere of application servers(assumed to belong to an ASP) that has been enriched with TAPAS features. A load generator has been used to exercise the application. The demonstration will show:

1. Electronic service SLAs contract monitoring: A third party service monitors the electronic service SLAs (specified in SLang) reporting any violations. Chapter 3 describes the implementation details of the QoS monitoring and violation detection service (implemented according to the architecture developed in TAPAS deliverable report D10) that performs this function. Integration with SLang for collection of metrics is also described.

2. Hosting SLAs contract monitoring: We have implemented adaptive clustering mechanism (QoS management, monitoring and Adaptation part shown in the TAPAS architecture) for incorporating QoS awareness into the application server. The demonstration will show the benefits of our clustering approach over the conventional clustering approach. The technical details of the clustering are described in a separate document (QoS-aware application server: preliminary design and implementation report); Chapter 4 presents the summary.

3. Terms and conditions contract monitoring: The inter-organisation interaction regulation system of the TAPAS architecture (TAPAS deliverable D9) ensures that only valid interactions (as defined in the contract) take place and non-repudiation information is automatically generated. The demonstration will show that activities such as bid placement generate non-repudiable evidence, and any non-contractual actions are not permitted. Chapter 5 describes these aspects.

4. Demonstration of QoS enabled group communication: The QoS enabled group communication system described in year two deliverable report, D8, has been integrated in the JBOSS application server. This integration is described in chapter 6. The auction application is not ideal for illustrating the features of the group communication system. We have therefore developed a separate application: a distributed collision detection system as required in distributed games/virtual reality applications. Chapter 7 describes the importance of having a QoS enabled group communication system for this purpose. A demonstration will be shown.

This set of demonstrations will illustrate all the important features of the TAPS platform.

2. Overview of the Auction Application

Werner Beckmann (Adesso) and Carlos Molina-Jimenez (Newcastle)

2.1. Introduction

As we expected, the auction application proved to have the broad variety of quality of service and trust related aspects that we needed to demonstrate most of the features of the TAPAS platform. This chapter describes the auction system and the various contracts involved.

2.2. Sealed reverse auctions

As discussed in [D13], from the point of view of the rules that dictate how bids are placed, there exist a large number of auction types (e.g. English auction, Dutch auction, sealed reverse auction, etc.); however, regardless of their type, all auctions involve the same components which interact with each other; for example, in all types of auctions we have an auctioneer who is responsible for running the auction house and who signs business contracts with its bidders.

Without loosing generality and motivated by commercial preferences of one of our project participant, in our demonstration scenario we consider a **sealed reverse auction** that is used by a car manufacturer (Toyota, Ford, etc.) for buying car parts (e.g. tyres, radiators, mirrors, etc.) from car part suppliers.

As explained in [D13], a sealed bid auction involves an auctioneer and a finite set of bidders ($B=\{bidder_1,\ldots,bidder_m\}$). The bidding process includes the opening and closing a finite number of bid rounds ($BR=\{BR_1,\ldots,BR_n\}$). The value of $n \geq 1$ is determined by the auctioneer and announced to all the bidders before opening the auction. Each bidder is allowed to place only one bid in each bid round. As the word "sealed" in the name of the auction implies, the value of a bid placed by a $bidder_i$ is known only to $bidder_i$ and to the auctioneer. The winner of each bid round is the bidder with the lowest bid; this explains why this auction is called reverse. The value of the winning bid is announced before opening the next round bid. A bidder can place a bid in BR_i ($2 \leq i \leq n$) only if he placed a bid in RB_{i-1}. A bid from a $bidder_i$ is accepted (considered placed) in BR_i ($2 \leq i \leq n$) only if the bid outbids the winning bid of RB_{i-1}.The winner of the auction is the winner of the last bid round (BR_m); however, an auction does not necessarily have a winner as the auctioneer reserves the right to declare the auction abandoned at any time during the bidding process.

Depending on the auctioneer's business preferences and resource availability, a reverse sealed bid auction can be deployed in the Internet differently; for example, we can think of an individual who uses his or her own technical resources (for example, his home computer)

to run his auction. However, in our demonstration scenario we have opted for generality, that is, we have thought of a general case where the auctioneer relies on other business partners to run his business. Figure [1] shown the parties that participate in our demonstration scenario. We will discuss the roles played by each party first and latter on we will discuss their contractual business relationships, which are represented by double-headed arrows in the figure.

- **The auctioneer** is car manufacturing organisation whose representatives run, at a given time, one or several instances of the auction with the purpose of buying car parts from car part suppliers; for example, he might run an instance of the auction for buying seats and another one for buying batteries. We assume that he does not want to be disturbed with computer-related issues, thus, he relies on somebody else to provide the infrastructure to run the auction; the business related activities which the auctioneer performs include selecting and sending invitations to potential bidders, opening and closing of bid rounds, declaring winners and so on.

- **The bidders** are car part manufacturing organisations interested in selling their products to the auctioneer. Naturally, each $bidder_i$ is represented by one or more representatives that participate in different auctions. In our scenario we consider that the total number of representatives can be in the order of several hundreds.

- **The auction application owner** is an organisation that offers the auctioneer the auction application ready to use; we can conceive it as an enterprise that owns a source code of the auction software and builds on-demand customized and ready to use copies of it to different auctioneers. The auction application owner is not involved in business related activities but in technical ones only; its responsibilities include running and tuning the auction software. We assume that the auction application owner does not have the ancillary resources to run his auction software, thus he relies on somebody else to host it.

- **The Application Service Provider (ASP)** is an organisation that offers hosting services to the auction application owner. It provides all the necessary ancillary resources (CPU, databases, ISP, human, etc.) to host the auction software. The assumption is that at a given time the ASP might be hosting several instances of the reverse sealed auction that belong to the one or several auctioneers as well as other applications of arbitrary nature.

- **The Internet Service Provider (ISP)** is an organisation that offers Internet connectivity to the APS so that the auction can be reached by other interested parties of the auction scenario. Notice that though it is not shown in Fig.[1], we assume that all the participants of the auction scenario are connected to an ISP.

- **The Storage Service Provider (SSP)** is an organisation that offers disk space to the ASP.

- **The business service** is an organisation that offers credit-rating services, or any other ancillary service, such as billing, to the auction application owner.

- **The Trusted Third Party (TTP)** is an organisation with enough credentials to act as a trusted third party. It measures the performance of a party, assesses it and determines whether the party is honouring its contractual obligations with respect to another party. For the sake of simplicity, Fig. [1] shows the TTP monitoring the contractual obligation only between the auctioneer and the auction application owner; however, a TTP can and should be deployed between any pair of business partners such as the auctioneer and bidder$_1$, and the auction application owner and the ASP.

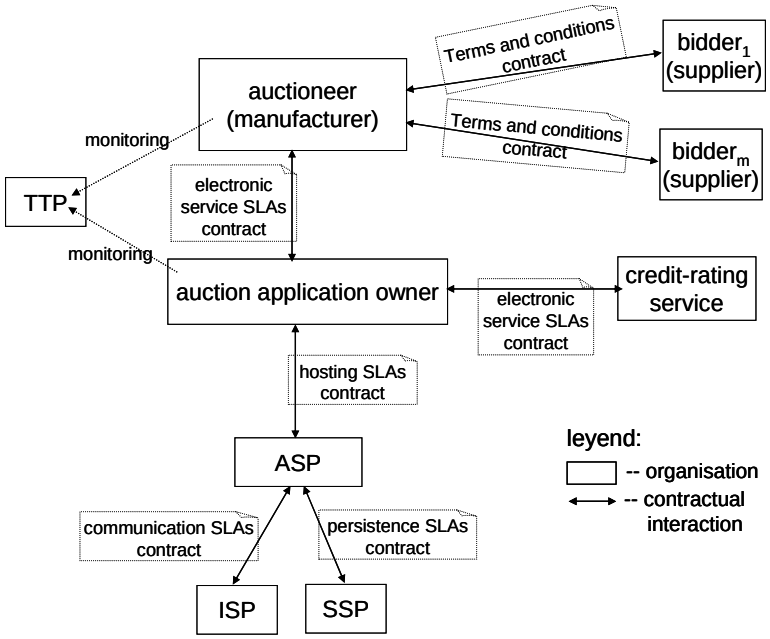


Fig 1. Auction scenario with its participants and their contractual relationships.

It is time now to discuss the meaning of the double-headed arrows that join the participants of our demonstration scenario shown in Fig [1]. A crucial assumption in our scenario is that the participants are autonomous and independent organizations that besides being mutually suspicious still want to conduct business together; because of the existence of this degree of mutual mistrust each pair of business partners needs to rely, as in conventional business, on legal business contracts to regulate their business interactions.

If in conventional business there are contracts for different commercial agreements (contracts for the rent of a house, contracts for loan of machinery, etc.) in our demonstration scenario, we identify five different types of contracts, namely, terms and conditions contracts, electronic service level agreement SLAs contracts, hosting SLAs contracts, communication SLAs contracts and persistence SLAs contracts. From a structural view, the five contract types are rather similar: all of them contain a header (signatories' names, addresses, signatures, start and end date, etc.) and clauses that stipulate the rights and obligations of each signatory party, however, from the point of view of the content of their clauses they are different:

- **Terms and conditions contracts** govern the relationship between the application owner and the bidders, i.e. the suppliers. In industrial scenarios users may alternatively be bound to terms and conditions issued by the auctioneer. Terms and

conditions are used to define standard relationships here, which do not contain individual agreements.

- **Electronic service SLAs contracts** stipulate the QoS expected from the interaction between the auctioneer/auction application owner pair and the auction application owner/credit rating service pair. For the first pair, the electronic service SLA will contain clauses such as "the auction application owner shall guarantee that even during peak periods the invocation of the place_bid operation is successfully completed within two seconds when there are less than 100 users logged in"; for the second pair the electronic service SLA will contain clauses such as "the auction application owner shall never place more that 50 request per second". In TAPAS, we have developed the language SLang for SLA specification.

- **Hosting SLAs contracts** are used to specify the QoS between the ASP and the application owner. They contain the objectives of the electronic service SLA because the application owner might prefer to delegate the objectives to other providers. Due to the more technically oriented relationship between application owner and ASP, the hosting SLA contains as well technical objectives such as memory space and utilisation regulations for technical services such as user management, maintenance windows etc.

- **Communication SLAs contracts** specify QoS objectives for the relationship between ASP and ISP. In today's industrial practice communication SLAs are quite well-known, though usually not specified in SLAng. Various tools exist to monitor certain aspects. In the current discussion, we therefore omit this type at present.

- **Persistence SLAs contracts** are used to specify QoS objectives for data storage service offered by an SSP. We do not focus on this relationship here

The distinction of different types of contracts is relevant because in our demonstration scenario contracts are meant to be represented, monitored and enforced electronically, that is, by means of computers and with little or none human intervention; from our research experience, we have learnt that the concepts and technology needed to represent, monitor and enforce a contract varies depending of the contract type. In our demonstration scenario we show how to represent, monitor and enforce only three types of contracts, namely, terms and condition contracts, electronic service level agreement contracts and hosting contracts; communication SLAs contracts and persistence contracts are left aside in our demonstration, our view is that the study of these two types of contracts needs some attention. In the following sections we discuss the contracts of our interest.

2. 3. Terms and conditions contracts

In previous section we briefly discussed that, in our auction demonstration scenario, the business interactions the business interactions that take place between the auctioneer and each bidder are regulated by what we called "terms and conditions". To be in accordance with legal terminology, it is necessary to clarify that these terms and conditions will appear as clauses in the **contracts signed between the auctioneer and its bidders, precisely, between the manager of two organisations involved** (the car manufacturing organisation

and the car part manufacturing organisation); naturally these clauses dictate the terms and the conditions under which a bidding company can gain access to the auction house. In this section we will discuss these contracts at large to explain how where they are located and how they are represented and enforced. It is worth clarifying that, in this document, we assume that contracts are already negotiated and signed.

Example of a contract between the auctioneer and a bidder

In this section we present the text of the contract that contains the terms and conditions that regulate the interactions between the auctioneer and his bidders. We assume that all the contracts that the auctioneer signs with his bidders contain the same clauses. The contract is written in English and though it is a hypothetical example, it contains terms and conditions that would be found in an actual contract. The conversion of the text contract into its electronic equivalent and its enforcement is discussed in Section [].

This deed of agreement (to be known as a Contract) is entered into as of the Effective Date identified below.

BETWEEN

Cars and Lorries Company, AG, Stockholmer Allee 24, Dortmund, 44629, Germany.

(To be known as the auctioneer in this agreement)

AND:

Goodgrip Tyres, Company, Fantasy Street, Newcastle, NE1 &RU, England.

(To be known as the bidder (or the bidding organisation) in this agreement)

WHEREAS the auctioneer agrees to provide the bidder with a selling auction service. The whole service is to be known as auction service in this document.

NOW IT IS HERBY AGREED that the auctioneer and the bidder enter into an agreement subject to the following terms and conditions:

1 Definitions and Interpretations

1.1 The execution of this agreement is to be performed by means of an electronic contract. Information exchanged between the auctioneer and the bidder, such as invitations to participate in an action, acceptance or rejection of invitations, bidding and notification is to be sent in electronic format.

1.2 A commitment deadline for an auction is to be understood as the time when a bidder that has received an invitation to bid in an auction has to accept or reject the invitation.

1.3 The auctioneer runs dynamic sealed reverse auction only.

1.4 The bidder shall gain access to the auction service from his standard browser; **installation of additional software in the bidder's computer shall not be required.**

1.5 In an auction, a bidder is represented by one or more representatives.

1.6 This agreement is signed under the British law; the auctioneer and the bidder hereby agree to observe the British jurisdiction during the execution of this contract.

2 Commencement and Completion

2.1 The commencement date is scheduled as 15 Sep 2004. On this date, the auctioneer and the bidder shall have their hardware and software infrastructure fully operational to conduct business.

2.2 The completion date is scheduled as 16 Sep 2005. Business operations invoked after this date shall be considered out of this contract.

2.3 Negotiation and signing of a new contract shall be required to continue the business relationship beyond the completion date.

3 Invitations to participate in actions

3.1 The auctioneer shall use his discretion to invite the bidder to participate in an auction. The auctioneer shall guarantee that invitations are received by the bidders at least 2 hours before the commitment deadline and not earlier than 4 hours before the commitment deadline.

3.2 The invitation shall specify two deadlines: commitment deadline and an auction opening time. The commitment deadline shall always be 2 hours before the auction opening time.

3.3 The auction opening time is to be understood as the time when the first bid round would start if the auction is announced opened.

3.4 The bidder shall at his sole discretion be entitled to accept or ignore the invitation to participate in an auction.

3.5 Acceptances of invitations shall be received at the auction server non-latter than the commitment deadline.

3.6 The auctioneer shall send a remind message to bidder that have failed to accept an invitation 1 hour before the commitment deadline.

4 Announcement of opening or cancellation of an auction

4.1 Auctions with two or more committed bidders shall be opened. Conversely, auctions with less that two committed bidders shall be cancelled.

4.2 The auctioneer shall guarantee that a committed bidder receives an announcement of either opening or cancellation, within 1 hour after the commitment deadline.

4.3 The auctioneer shall include in his auction opening announcements three parameters: the number of bid rounds (n), the duration of the bid rounds (t_{br}) and the length of the time interval between the bid rounds (t_{brg}). The selection of the value of these parameters shall be the sole responsibility of the auctioneer.

4.4 Bid rounds shall be identified as BR_1, BR_2,…,BR_n, where n is the number of bid rounds announced by the auctioneer when the auction was declared opened.

5 Expel from an auction

5.1 The auctioneer shall reserve the right to expel an invited bidder from an auction if the auctioneer is not satisfied with the bidder's behaviour or with the bidder's financial credibility.

5.2 The auctioneer shall use his discretion to expel an unwanted bidder:

1- At any time between the opening and closing of RB_i ($1 \le i \le n$).

2- The auctioneer shall notify the expelled bidder about his expelling before the announcement of the winning bid of the current bid round.

6 Bidding

7 The bidder has the right to have one or more representatives placing bids in an auction; however, at bidding time, the auctioneer shall not distinguish between two ore more representatives belonging to the same bidder.

7.1 An invited bidder shall have the right to bid in the auction by means of place_bid operations send to the auctioneer.

7.2 The opening of BR_1 shall be understood as the auction opening time (see 3.3).

7.3 The opening time for a BR_i ($2 \le i \le n$) is determined by requirements specified in 4.3 and shall be announced by the display of the winning bid of BR_{i-1}.

7.4 The auctioneer shall guarantee that a winning bid is announced 5 minutes after closing the bid round.

7.5 An invited bidder is allowed to place a bid only if the bid round is opened and the bid satisfies the following requirements:

1. The bidder can place a bid in a given bid round BR_i ($1 \le i \le n$) only if the bidder does not have an accepted bid in BR_i.
2. The bidder can place a bid in a given bid round BR_i ($2 \le i \le n$) only if he placed a bid in bid round RB_{i-1}.
3. A bid is accepted in bid round BR_i ($2 \le i \le n$) only if the bid outbids the winning bid of RB_{i-1} by 10%.
4. The auctioneer shall acknowledge the acceptance or rejection of a bid within 3 minutes after receiving the bid.

8 Auction outcome

8.1 The auctioneer is entitled to declare the auction abandoned if he is not satisfied with the outcome of the last bid round.

8.2 The auctioneer shall notify (by e-mail) the outcome (either "auction abandoned" or "winning bidder with bid-value") of the last bid round within 5 minutes after closing the bid round.

8.3 A winning bidder is in the obligation to sell his item or items.

8.4 A winning bidder shall contact the auctioneer within 10 minutes after the declaration of the winning bidder.

9 Payment and delivery

9.1 Arrangement for payment and delivery are to be conducted outside the electronic auction system, for example over the phone.

2.4. Monitoring and enforcing terms and conditions contracts

To explain where a contract that contains terms and conditions fit in our auction scenario it might be helpful to discuss Fig. [3]; this figure abstracts away all the unnecessary complexity to show only the elements that matter for this discussion. In the figure, the auction server represents the infrastructure (hardware and software equipment) where the auctioneer's auction is deployed; a set of $m \geq 1$ bidders use their personal computer to access the auction application through the Internet Service Provider ISP_i; similarly, the auctioneer relies on ISP_i to run his auction from the comfort of his home computer; the interaction between the auctioneer and the bidders is strictly regulated by legal contracts, since we assume that the bidders are business-independent from each other, the figure shows $m \geq 1$ contracts, signed between the auctioneer and each bidder. As suggested by the figure, a contract is conceptually located between its two signatories and it is there to enforce that the rights and obligations derived from the terms and agreements are honoured or their violation detected and notified when the signatories interact with each other. In practice auctioneers offer their services on the "take it or leave" basis, this means that the terms and conditions stipulated in these contracts would not be negotiable; if this assumption is correct, these contracts would have identical terms and conditions.

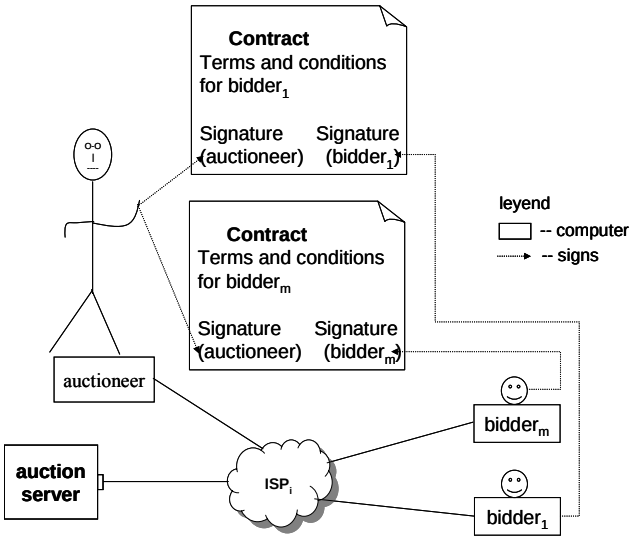


Fig. 3 Contractual terms and conditions between the auctioneer and his bidders.

Representation of the contract with FMSs

The contract shown in Section 2.3 can be represented as a FSM as shown in Fig. 4a, 4b and 4c. When converted into an executable code, the FSM contract will ensure that the auctioneer and the bidder honour what the text contract stipulates. For instance it will ensure that the timing requirements such as invitations to be sent non-later than two hours before the commitment deadline are met (see Fig. 5).

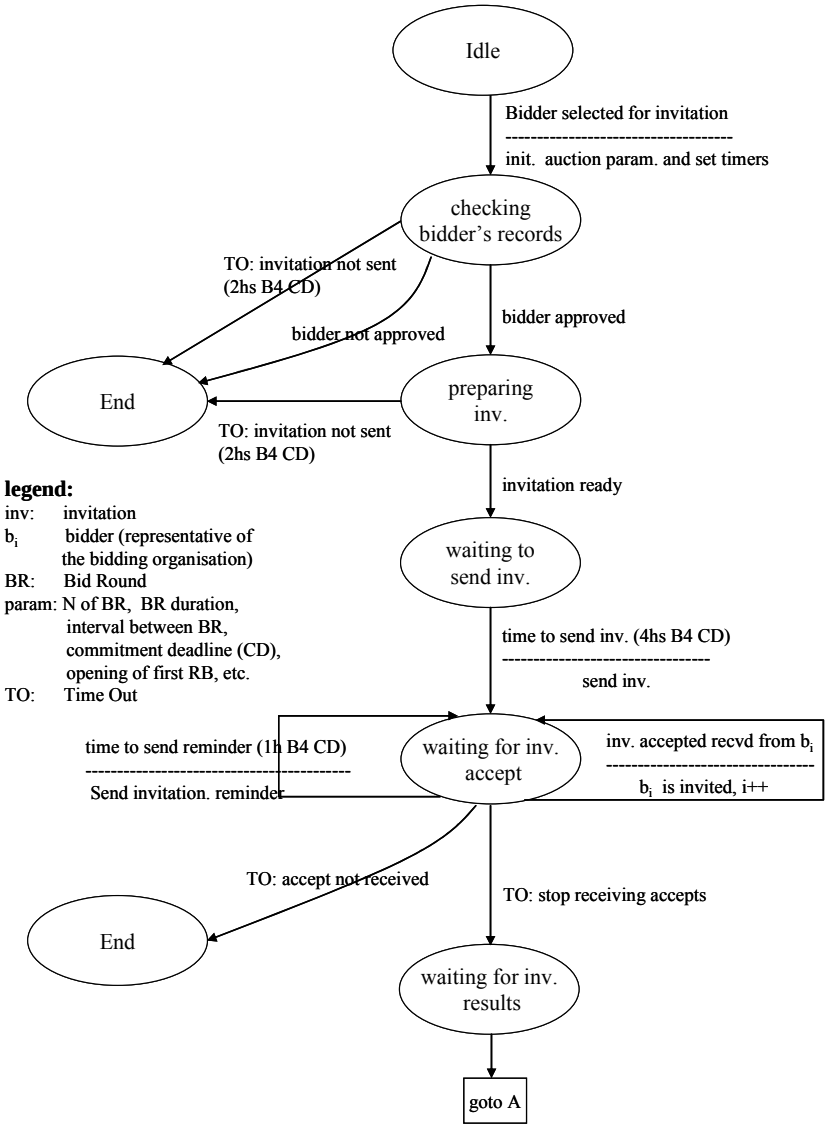


Fig. 4a Representation of a contract by means of FSMs.

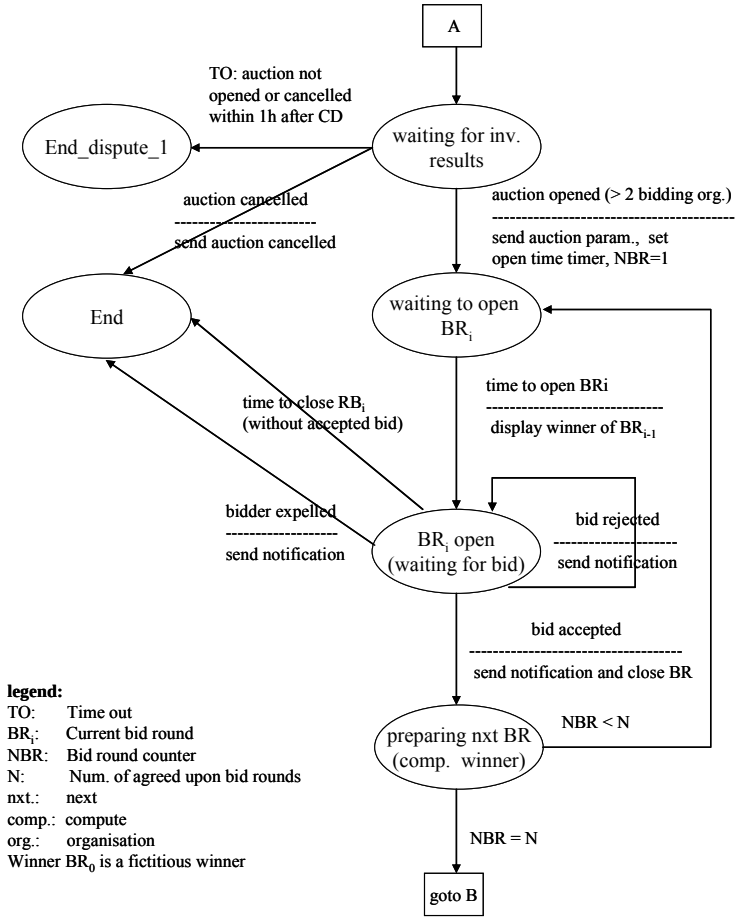


Fig. 4b Representation of a contract by means of FSMs (cont.).

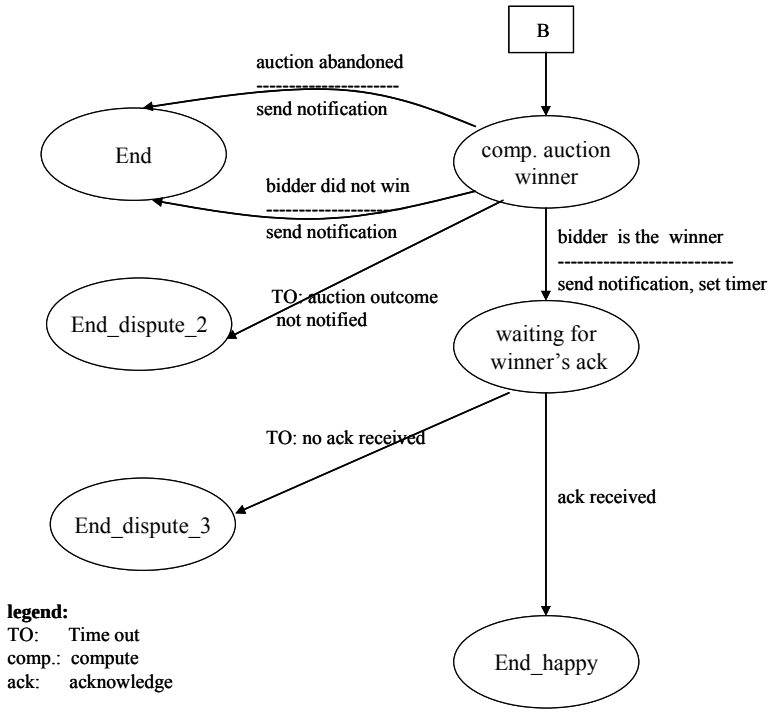


Fig. 4c Representation of a contract by means of FSMs (cont.).

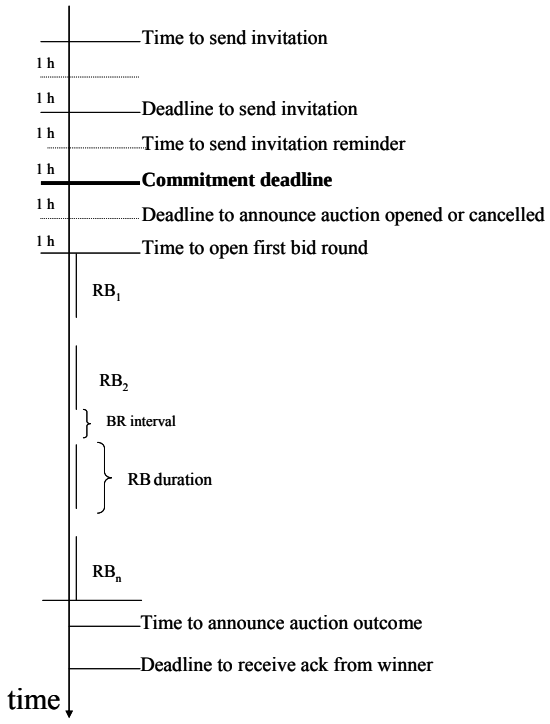


Fig. 5 Timing requirements.

2.5. Electronic service SLAs

We describe a typical contract between the auctioneer and the owner of the auction application.

This deed of agreement (to be known as a Contract) is entered into as of the Effective Date identified below.

BETWEEN

Auction Services, AG, Stockholmer Allee 24, Dortmund, 44629, Germany.

(To be known as the auction application owner)

AND:

Cars and Lorries Company, AG, Stockholmer Allee 24, Dortmund, 44629, Germany.

(To be known as the auctioneer in this agreement)

WHEREAS the auction application owner agrees to provide the auctioneer with a customized and ready to use software for running reverse sealed auctions. The whole service is to be known as auction service in this document.

NOW IT IS HERBY AGREED that the auctioneer and the auction application owner enter into an agreement subject to the following terms and conditions:

1. Definitions and Interpretations.

1. The execution of this agreement is to be performed by means of an electronic contract. Information exchanged between the auctioneer and the bidder, such as invitations to participate in an action, acceptance or rejection of invitations, bidding and notification is to be sent in electronic format.

2. A commitment deadline for an auction is to be understood as the time when a bidder that has received an invitation to bid in an auction has to accept or reject the invitation.

3. The auctioneer runs dynamic sealed reverse auction only.

4. The bidder shall gain access to the auction service from his standard browser; **installation of additional software in the bidder's computer shall not be required.**

5. This agreement is signed under the British law; the auctioneer and the bidder hereby agree to observe the British jurisdiction during the execution of this contract.

2. Commencement and Completion

1. The commencement date is scheduled as 10 Aug 2004. On this date, the auction application owner shall have its software infrastructure fully operational to conduct business.

2. The completion date is scheduled as 30 Nov 2005. Business operations invoked by the auctioneer after this date shall be considered out of this contract.

3. Negotiation and signing of a new contract shall be required to continue the business relationship beyond the completion date.

3. Service level parameter

1. The owner of the auction application shall provide a service that can run at least 100 auctions simultaneously.

2. The owner of the auction application shall provide a service that can log in at least 50 bidders per auction, simultaneously.

3. The owner of the auction application shall guarantee a response time of less than 5 seconds for a place_bid operation.

4. The owner of the auction application shall provide a fair announce service which shall guarantee that announcements will be received by all active bidders within a time interval of 3 seconds.

5. The owner of the auction application shall provide allowances for bidders' thinking time.

5.1. "bid_round_closing" announces shall be received by active bidders at least 5 minutes before the actual closing time.

2.6. Monitoring electronic service SLAs contracts

Fig 6 shows the architecture that we use for monitoring and enforcing the electronic service SLAs contract that regulates the interaction between the auctioneer and the auction application owner of our demonstration scenario (see Fig. [1]). This architecture comes from TAPAS deliverable D10.

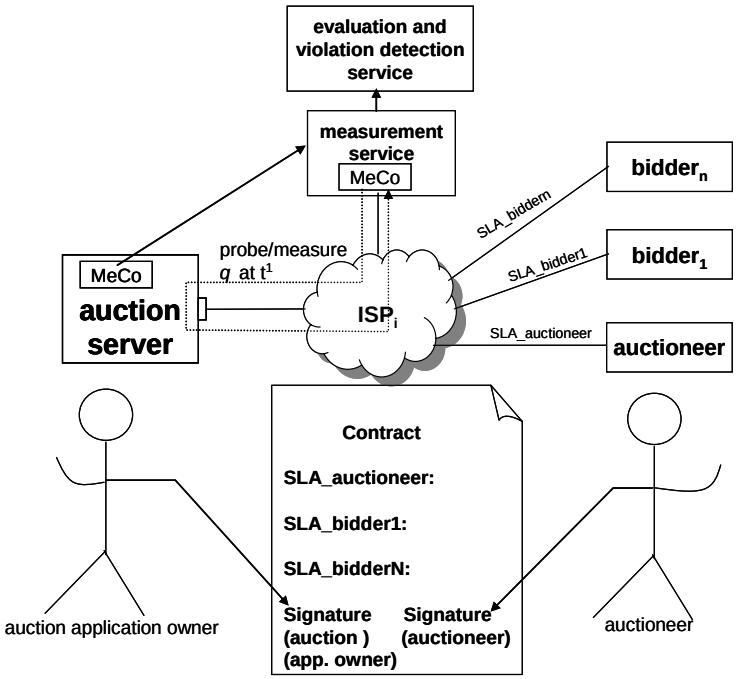


Fig. 6 An electronic service SLAs contract between the auctioneer and the auction application owner.

In the figure we assume that the auction server is remotely accessed by the auctioneer to manage his auction business and by bidders ($bidder_1$,…,$bidder_n$) interested in the auction services offered by the auctioneer. The quality of the service (QoS) delivered by the auction server as seen by the auctioneer and the bidders is stipulated in terms of electronic service level agreements. As shown in the figure, this contract is signed by the auction application owner and the auctioneer. Notice that for generality the contract contains individual service level agreements (one for each bidder). It is sensible to think that in practice we will have a single electronic service level agreement for all the bidders. Chapter 3 describes the implementation.

2.7. Hosting SLAs contracts

A hosting SLA contract will be similar to the electronic service SLAs; most of its QoS parameters will be taken from the electronic service SLAs. In addition, this hosting SLAs contract will contain additional details that are implementation specific (e.g., storage requirement). The hosting SLAs contract is used by the TAPAS QOS -aware application server for determining its configuration. This aspect is described in chapter 4.

References

[1] Werner Beckmann and Sabine Lembke, *Use Cases of the Auction Platform*, 1 Apr 2001.

[2] W. Beckmann and M. Kossmann and S. Kramlowsky and U. Radmacher; *Second year evaluation and assessment report (D13)*, Adesso AG May 25th 2004.

3. Integrating SLA monitoring with SLAng

Graham Morgan, Simon Parkin (Newcastle) and James Skene (UCL)

3.1. Introduction

Our approach for monitoring electronic service SLAS contract is based on our earlier work described in [16] and work associated to SLA specification and evaluation carried out at University College London (UCL) [14].

Our earlier work [16] discusses the fundamental issues that monitoring of this kind of contractual obligations involves: SLA specification, separation of the computation and communication infrastructure of the provider, service points of presence, metric collection approaches, measurement service and evaluation and violation detection service. With consideration of the issues associated with monitoring electronic service SLAs contracts, [16] develops an architecture that is the basis for the implementation presented here (shown in Figure 1).

Work carried out by UCL [17] describes the motivation for performance analysis in the domain of Enterprise Information Systems (EISs) and argues that the Model Driven Architecture (MDA) is a suitable framework for integrating formal analysis techniques with engineering methods appropriate to the domain. The MDA permits natural and economical modelling of design and analysis domains and the relationships between them, supporting both manual and automatic analysis. Unified Modelling Language (UML) is extensively used to capture system designs and presents a general modelling approach and outlines its use in relating models of applications, annotated using standard profiles, to analysable formal models. The result of this work is a language that may be used to specify SLAs and an implementation of an SLA checker. The monitoring system we have constructed uses metric collection as defined in the SLA language developed at UCL and uses the SLA checker for SLA evaluation.

We assume inter-organisational communication is enacted over the Internet using SOAP. In order to prevent the minimum of interruption to existing service provider/consumer interaction, the implementation of our SLA-monitoring framework is achieved via technologies that require no alteration to application logic at the service provider and consumer.

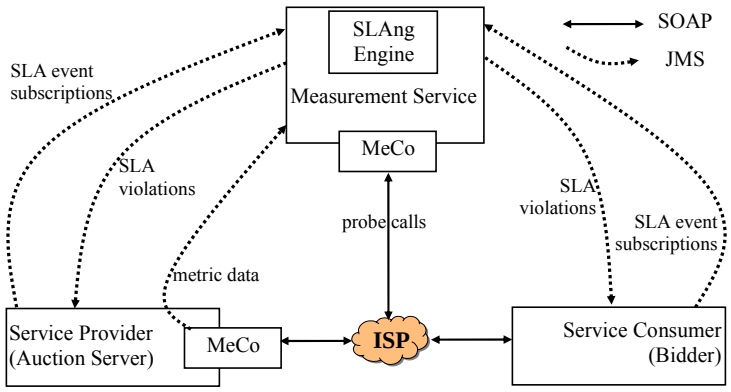


Figure 1 – Electronic service SLAs monitoring architecture.

We use an online auction as our example application (i.e., service providers are auctioneers and service consumers are bidders). The architecture shown in figure 1 extends the typical provider/consumer relationship with a number of additional components required to implement our monitoring framework:

- **Service provider MeCo** - This MeCo (metric collector) intercepts service consumer requests (and associated outgoing responses) and records measurements based upon a service consumer's usage of the service provider's platform. These measurements aid in determining if a service consumer is violating an SLA by using a service inappropriately.

- **Measurement service MeCo** – This MeCo observes the performance of service providers by assuming the role of a service consumer. Periodic probing of the service provider is enacted by the measurement service MeCo to gain measurements relating to the performance of a service provider as viewed by a service consumer. These measurements aid in determining if a service provider is satisfying service consumers as specified in an SLA.

- **Measurement service** – Responsible for collating the measurements gathered from MeCos and informing SLA participants of SLA violation

- **SLAng engine** – A sub-system of the measurement service that is responsible for detecting SLA violations given metric data supplied by the measurement service.

- **Messaging service** – Provides communication platform across which metric data and SLA violation notifications are propagated throughout the system.

The measurement service may be within the domain of a trusted third party, ensuring that service provider and consumer may abide by the decisions on SLA violations generated by the SLAng engine.

In the following sections we describe the implementation of each component and how different components collaborate to provide SLA monitoring and notification.

3.2. Metric Collectors (MeCos)

MeCos are responsible for gathering metric data and propagating such data to the measurement service for evaluation. Service providers have a MeCo within their organisational domain for monitoring service consumer usage. MeCos are suitable for use with arbitrary protocols. Different protocols may be supported with the use of *MeCo hooks*. MeCo hooks are protocol dependent and are responsible for the interception of consumer request/replies messages and passing such messages through the MeCo. MeCo hooks are provided by our system for supporting SOAP and Java Remote Method Invocation (suitable for Enterprise Java Beans invocation). As we assume communications between service provider and consumer are SOAP based we shall restrict our discussion to the MeCo SOAP hook.

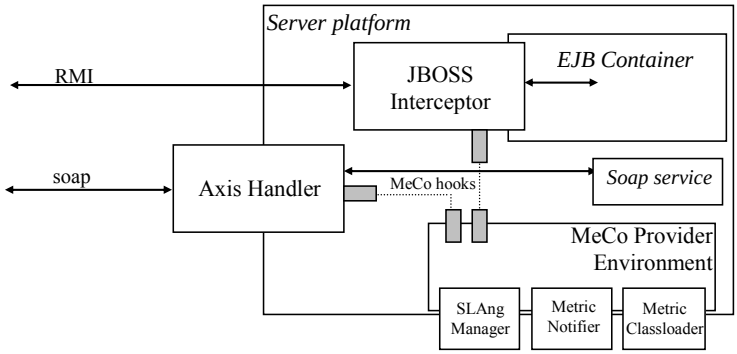


Figure 2 – Service Provider use of MeCos.

Our SOAP implementation is provided by Implementation Apache eXtensible Interaction System (Axis) [18]. Axis provides handlers (*Axis Handlers*) that may be chained together to provide a mechanism for interception, and possible alteration of a SOAP message (e.g., add/remove headers, manipulate the body), at different points during traversal of the protocol stack (i.e., before request is processed by server side logic or before reply is received by a client). Axis handlers provide an appropriate opportunity to redirect SOAP messages to a MeCo (via MeCo hooks) for metric gathering. The addition of Axis handlers does not require alterations to the application logic, therefore the introduction of monitoring at the service provider may be achieved in a transparent manner.

Figure 2 shows the architecture of MeCo deployment in the service provider. Both EJB and SOAP MeCo hooks are shown for completeness. However, in the demonstrator auction application only MeCo SOAP hooks are used. In addition to the assumption that that inter-organisational communications is more appropriately achieved using SOAP, we consider the possibility of a clustered computer scenario for server side scalability. We assume load balancing occurs after an Axis handler has intercepted a SOAP message. Therefore, cluster management and possible redirection of consumer requests will not influence the metrics gathered by the service provider MeCo in an undesirable manner. This is an important consideration as the redirection of a consumer request may result in up to twice the expected processing to be witnessed by a MeCo installed on one of the cluster nodes. For example,

consider client request M_1 that is received at node N_1. If node N_1 fails to satisfy the request (possibly due to node failure) then a load balancer may redirect M_1 to N_2 to ensure the service provider may satisfy M_2. If MeCo deployment occurs at the nodes, then two sets of metric data will be gathered when the consumer has, in reality, only sent a single request. In the worst case scenario this may be flagged as an over usage of service resources by a consumer and result in SLA violation.

The MeCo provider environment contains a number of components that cumulatively satisfy the metric collection and dissemination (back to the measurement service) requirements of our monitoring system (shown in figure 2):

- **SLAng Manager** – Examines an SLA contract file (as used by SLAng engine) to determine the metric data that the MeCo is to observe. As there may be many SLAs that a MeCo is responsible for monitoring, streamlining of the monitoring may occur by avoiding duplicate monitoring requests. For example, if SLA_1 and SLA_2 describe the upper bound latency for a client invocation C_1, then the message interception associated to C_1 by a single MeCo hook may satisfy the monitoring requirements of both SLA_1 and SLA_1.

- **Metric Notifier** – Based on the deduction of what to monitor made by the SLAng manager, the metric notifier assumes responsibility for managing the appropriate message passing between MeCo and measurement service. This requires the creation of the appropriate message channels over which metric data will travel.

- **Metric Classloader** – Loads the appropriate classes that are required for implementing the monitoring of the required data as specified by the SLAng manager. Classes are loaded from the class repository. Each class represents a metric type as specified by the SLA language used by the SLAng engine (e.g., response time).

The MeCo provider environment was developed in a modular fashion so the minimum of tailoring was required to make the MeCo work with different protocols, different application types and different SLA languages. The MeCo hooks, as already discussed, allow different protocols to be supported. For each SLA language a different SLAng manager and class repository is required. This is because such a language must be parsed (by the SLAng manager) and appropriate mechanisms for metric data monitoring realised (by class repository). This approach has the added benefit of allowing our system to be extendable in that any extensions that may be added to an SLA language over time may be incorporated into the MeCo.

The MeCo in the measurement service differs from the MeCo located in the service provider in that the measurement service MeCo is employed to periodically probe the service provider. Probing in this manner is carried out to gain metric data relating to how service provider appears to be performing as viewed by a service consumer (e.g., response time of service provider). The MeCo may be configured manually to carry out the desired probing. Metric data generated by the measurement service is handled in the same manner as the metric data generated at the service provider MeCo.

3.3. Messaging Service

The messaging service is responsible for passing metric data from the service provider MeCo to the measurement service and passing SLA violation detection messages from the measurement service to interested parties of an SLA. Message Oriented Middleware (MOM) (Java Messaging Service – JMS) was chosen as the message platform.

JMS provides a standard API that allows Java developers to integrate MOM into their applications. The JMS specification does not indicate how the underlying system implementation is achieved (e.g., architecture) resulting in a number of varying solutions available from different vendors. JMS supports point-to-point and publish/subscribe models of interaction. Point-to-point is based on the notion of queues, with a queue identified as an asynchronous mechanism for passing messages from suppliers to consumers. A client may get all its messages delivered to a queue, allowing a queue to contain a variety of different message types. Publish/subscribe is based on topics, with clients publishing and subscribing to well defined topics. The topic acts as a mechanism for gathering and distributing related messages (as perceived by an application) to clients and allows subscribers and publishers to be unaware of each other's existence.

The choice of JMS as opposed to other messaging approaches (e.g., Java RMI) was due to the ease with which the desired message dissemination may be achieved throughout our system. The topic approach was chosen with the measurement service creating a topic on a per operation basis (e.g., the name of a method associated to an operation). We call such topics *metric topics*. The measurement service consumes messages as and when they are published on the metric topics. This is a desirable scenario when we consider a large scale deployment of our monitoring architecture. Assuming we have multiple service providers, there is no need for each service provider MeCo to create a direct communication channel to the messaging service. Requiring a messaging service to manage communication links to hundreds or thousands of service providers is not scalable. A service provider MeCo starts disseminating metric data by publishing such data on an appropriately named metric topic. This also provides opportunity to provide multiple measurement services.

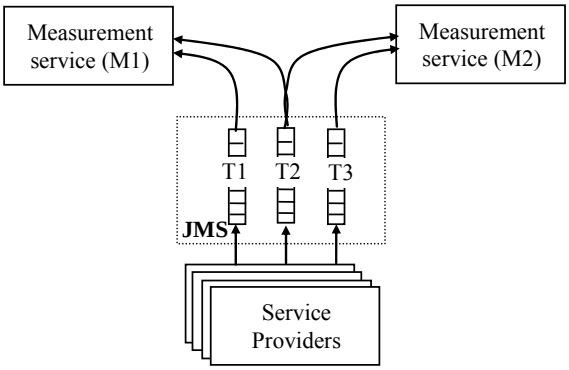


Figure 3 – Message scenario using metric topics in JMS.

Figure 3 illustrates a possible scenario with multiple service provider MeCos publishing metric data on metric topics T_1, T_2 and T_3. Measurement services M_1 and M_2 are consuming metric data from these metric topics and are responsible for identifying SLA violation. The set of SLAs M_1 is responsible for is different than the set of SLAs M_2 is responsible for, allowing M_1 and M_2 to share the processing load associated to SLA violation. The introduction of additional measurement services in this manner is straightforward: a measurement service registers as a consumer for the metric data they are interested in (to enable SLA violation detection). There is no need to contact each service provider MeCo directly in order to retrieve metric data (a costly process if service providers are measured in their hundreds or thousands), allowing additional measurement services to be added with minimum disruption to the overall function of the system. This approach also benefits from an ability to support multiple third party measurement services. A scenario may exist where a service provider may provide services to multiple service consumers, with such consumers requiring different third parties to govern their SLA violation detection mechanisms.

A metric topic message is dependent on the SLA language used. To accommodate the SLAang engine [17] we used to implement our service, a metric topic message contains the metric ID (unique identifier associated to a particular metric), values monitored (metric data), client ID (unique identifier associated to service consumer) and server ID (unique identifier associated to service provider). The metric ID is a derived from the method invocation (defined is WSDL) associated to the SOAP message and the client/server IDs are the IP addresses associated with service consumer and provider respectively.

Propagating an SLA violation to SLA participants is achieved via a JMS topic (*SLA topics*). Such topics are created on a per SLA basis, with organisations assuming responsibility for registering as subscribers on the SLA topics that govern their inter-organisational contractual obligations. The subscription of organisations to SLA topics and the actions taken by organisations that receive SLA violations is beyond the scope of our system (we are only interested in monitoring). A SLA topic message consists of a metric ID (associated to the metric that was violated).

3.4. Measurement Service

The measurement service evaluates metric messages received from metric topics and notifies organisations, via SLA topics, of SLA violations. The measurement service contains a number of components (figure 4):

- **SLAng Message Manager** – Examines an SLA and determines which metric and SLA topics are required. Metric and SLA topics are created when required by the SLAng message manager. In addition, when an SLA is withdrawn from use the SLAng message manager deletes the appropriate SLA and metric topics (after determining that the metric topics flagged for deletion are no longer required by other, active, SLAs).

- **Metric Listener** – Subscribes to the appropriate metric topics as instructed by the SLAng message manager and assumes responsibility for consuming metric topic

messages and translating such messages to a format suitable for acceptance by the SLAng engine.

- **SLAng engine** – Receives messages from the metric listener and issues SLA violation notification messages.
- **Violation Notifier** – Subscribes to the appropriate SLA topics as instructed by the SLAng message manager and assumes responsibility for translating violation notification messages received from the SLAng engine to JMS messages and issuing such messages on SLA topics.

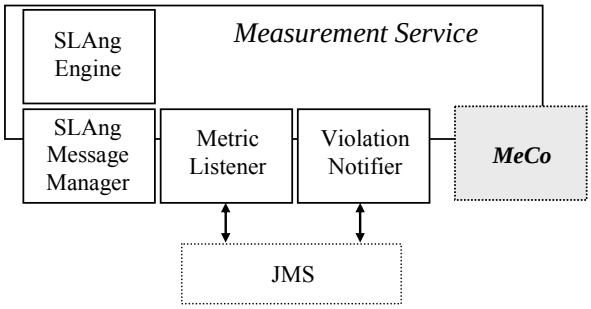


Figure 4 – Measurement service.

The SLAng message manager is require when new SLAs are introduced or existing SLAs are deleted and can be viewed as carrying out the tasks to maintain the required SLA and metric topic subscriptions of the metric listener and violation notifier. The metric listener must translate the metric data it receives from metric topics into a suitable format for submission to the SLAng engine. For our purposes this required a *service usage* message to be created. A service usage message is a description of how a service was used, based on duration of use, operation use, identified service consumer and the actual time of entry of the client request into the service provider platform. The SLAng engine examines service usage messages to determine if SLA violation has occurred or if service usage has been enacted within acceptable bounds. The SLAng engine informs the violation notifier of SLA violations, indicating which SLA has been violated (allowing the violation notifier to realise which SLA topic to issue a violation message). The SLAng engine includes in a SLA violation message the part(s) of the SLA (relating to specific metrics in the SLA contract) which have been violated. The violation notifier includes in the violation message details relating to what caused the SLA violation in the message issued to the appropriate SLA topic.

3.5. Related Work

A number of Quality of Service (QoS) management frameworks already exist, both within research communities and in commercial environments. They provide a range of services covering topics such as the specification of QoS requirements, the allocation of resources to meet such requirements, and the monitoring of system performance to determine whether QoS requirements are being satisfied. A desirable property in such frameworks is that QoS management must demand as little modification to (and preferably as little run-time

interference of) the system being monitored as is possible. A way of achieving this is by way of specialised QoS middleware [5, 8]. An approach for generalised QoS provision via the deployment of specialised middleware is described in [8]. The notion of 'interceptors' (middleware components that can be placed between application components to provide further functionality) is used to propose an infrastructure of re-usable QoS components which can be inserted at both the client and server sides of an observed QoS system, with the intention of clearly separating QoS provision from application development. Furthermore, these interceptors can be dynamically added in chains between the application and any existing middleware. QoS-oriented examples of such interceptors have already been implemented (as demonstrated in [8]), including failover support and admission control.

The Quality Objects (QuO) project [5] describes a CORBA-based QoS-oriented gateway mechanism, inserted at the transport layer between communicating processes, that provides an extensible framework to allow for the transparent insertion of QoS control mechanisms at a fixed point in the communication stream. This centralised point of management acts as an integrated interface for the management of QoS control properties (and their related programmatic mechanisms), while also overseeing the collection of information relating to behaviour of the system and the QoS demands of the QuO platform itself, for use in the management of particular aspects of QoS provision. With this, a conceptual language for the representation of QoS information (referred to as Quality Description Languages, or QDL) was also developed, to assist in the definition of service contracts and measurement data selection. This combination of centralised QoS provision and integrated QoS measurement together in a gateway object provides a transparent mechanism for the definition of QoS requirements, the management of QoS-related information, and the provision of QoS-oriented mechanisms within an existing system.

Some systems exist that provide a system-wide QoS framework, which can be actively deployed in order to satisfy the QoS requirements of a service and associated clients. The WSLA Framework [4], developed by IBM to address the need for automated SLA monitoring and enforcement for Web Services where service relationships may be created or absolved arbitrarily and in a dynamic fashion. Within this project, various types of system metrics are identified and put into context with relation to both service consumers and providers of associated services, so as to better understand their position within the management of QoS properties. Furthermore, the WSLA Framework defines a rigid SLA lifecycle, around which a monitoring infrastructure has been developed; this lifecycle describes the initial negotiation of SLAs, as well as the monitoring of QoS parameters as stipulated in the agreement, and the subsequent evaluation of these parameters with respect to the terms of the governing SLA, coupled with a notification mechanism for the distribution of SLA violations (with respect to the parties directly involved in the observed system). As mentioned, these facilities are embodied in a monitoring infrastructure, which is primarily driven by the SLA Compliance Monitor, which houses services for the maintenance of system configurations and the measurement of data pertaining to run-time system behaviour – this measurement is achieved either by way of mechanisms internal to a service provider (which afford direct access to provider-side metrics), or through interception (or alternatively active replication) of client requests. In addition to the core SLA Compliance Monitor, there is provision within the WSLA framework for the delegation of

monitoring tasks to trusted third parties, for those situations where involved parties wish to involve an impartial QoS measurement body.

As well as the broader QoS management infrastructures, there are more specialised QoS monitoring frameworks, which focus primarily on the monitoring and reporting of QoS metrics. In [6] an automated SLA monitoring infrastructure for use in a Web Services environment is proposed. Proxy components are deployed within the SOAP communications layer with the purpose of intercepting messages as they pass between communicating parties, as well as also examining the internal logs of relevant software components where available (in order to correlate intercepted messages with recorded business process activity). These actions are correlated by a Business Management Platform (BMP), which dynamically manages SLAs (with respect to monitoring and violation detection/notification).

A CORBA-based QoS monitoring system which focuses on network routers and traffic engineering is presented in [7], with a view to providing a monitoring infrastructure that is scalable (with respect to the number of network nodes and the speed of the underlying network, as well as the number of clients utilising the value-added services that are being observed). The system that was developed incorporates numerous novel approaches to achieving scalability, such as the minimisation of cross-component notification overheads, as well as reduction of synthetic traffic as part of the active monitoring process, and controlling levels of synthetic traffic by balancing the injection rate with the resultant network load. Node Monitors (hosted on machines outside the network routers) perform passive measurements on their associated routers, and active measurements to coordinate with other node monitors (on a path-level or per-hop basis), as well as limited evaluation of gathered metrics. Non-real-time processing of the measurement data is carried out by the global Network Monitor, which receives and correlates data from the node monitors. Both the network monitor and the node monitors are directly connected to the Service Level Specification (SLS) Monitor, which observes and reports SLA-related events. The entire monitoring system is augmented by a monitoring repository, which is used to store all of the collected data, along with limited amounts of system configuration data.

There exist commercial QoS monitoring systems, such as those provided by Keynote Systems [9]. Keynote's enterprise-scale systems are capable of utilising a number of data collection methods, such as the examination of server platform logs (in order to build a detailed representation of service behaviour), which can then be augmented by active measurement mechanisms which are capable of probing a provider platform from a specific point within the Internet (via a wide range of connection technologies) within a specified monitoring schedule, in order to provide a user-oriented context for the measurement data gathered from the observed provider. All of the data collected can be correlated via service-tracking mechanisms, which can in turn be employed to determine whether SLA conditions are being upheld within the observed system. The various methodologies employed by Keynote's various enterprise-scale monitoring infrastructures are coordinated on-demand, in an attempt to build an accurate representation of system behaviour and allow for the comparison of the correlated system view with the contractual expectations of the associated service provider.

In light of the work described here, there is still the need to provide a non-intrusive SLA-monitoring framework that can be easily deployed within an existing system that utilises already-available enterprise-scale application technology (such as EJB applications or SOAP-based Web services) in a transparent fashion, while still affording accurate measurement of system behaviour characteristics and automated evaluation of measurement data against an electronic SLA contract. With this premise in mind, we developed the Metric Collector (MeCo) framework.

References

[1] Markus Debusmann, Alexander Keller, "SLA-Driven Management of Distributed Systems Using the Common Information Model", in Proceedings of the 8th IFIP/IEEE IM, 2003

[2] Chris Overton, "On the Theory and Practice of Internet SLAs", Journal of Computer Resource Measurement 106, 32-45, Computer Measurement Group, 2002

[3] Ahsan Habib, Sonia Fahmy, Srivinas R. Avasarala, Venkatesh Prabhakar, Bharat Bhargava, "On Detecting Service Violations and Bandwidth Theft in QoS Network Domains", Computer Communications, Elsevier, Vol. 26 Issue 8, Pages 861-871, 2003

[4] Alexander Keller, Heiko Ludwig, "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services", IBM Research Report, 2002

[5] Richard Schantz, John Zinky, David Karr, David Bakken, James Megquier, Joseph Loyall, "An Object-Level Gateway Supporting Integrated-Property Quality of Service", ISORC '99, 1999

[6] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Li Jie Jin, Fabio Casati, "Automated SLA Monitoring for Web Services", HP-Labs Report HPL-2002-191, 2002

[7] Abolghasem Asgari, Panos Trimintzios, Mark Irons, Richard Egan, George Pavlou, "Building Quality-of-Service Monitoring Systems for Traffic Engineering and Service Management", Journal of Network and Systems Management, Vol. 11, No. 4, 2003

[8] Jim Pruyne, "Enabling QoS via Interception in Middleware", HP-Labs Report HPL-2000-29, February 2000

[9] Keynote Systems, http://www.keynote.com

[10] DevelopMentor, International Business Machines Corporation, Lotus Development Corporation, Microsoft, UserLand Software, "Simple Object Access Protocol (SOAP) 1.1", http://www.w3.org/TR/SOAP/

[11] Axis Toolkit, http://ws.apache.org/axis

[12] Sun Microsystems, Enterprise JavaBeans Specification, Version 2.1, 2003, http://java.sun.com

[13] JBoss, http://www.jboss.org

[14] James Skene, D. Davide Lamanna, Wolfgang Emmerich, “Precise Service Level Agreements”, Proceedings of the 26th International Conference on Software Engineering, Pg. 179 – 188, 2004

[15] Sun Microsystems, Java Message Service (JMS) Specification, http://java.sun.com/products/jms, Version 1.1, 2002

[16] C. Molina-Jimenez, S. Shrivastava, J. Crowcroft, and P. Gevros, “On the Monitoring of Contractual Service Level Agreements”, In Proceedings of the IEEE Conference on Electronic Commerce CEC\04, The First IEEE International Workshop on Electronic Contracting (WEC), San Diego, 6-9, 2004 (also, TAPAS deliverable, D10).

[17] J. Skene and W. Emmerich (2003). Model Driven Performance Analysis of Enterprise Information Systems. Electronic Notes in Theoretical Computer Science, 82(6).

[18] R. Irani, S. J. Basha, “AXIS: Next Generation Java SOAP”, Peer Information; 1st edition, 2002.

4. Hosting SLAs, a story board

Giorgia Lodi, Fabio Panzieri (Bologna)

The purpose of the storyboard two is to show the functionalities of the TAPAS QoS-enabled application server, which is deployed in a cluster of workstations.

Figure 1 below depicts the scenario that demonstrates the TAPAS clustering benefits.

The environment consists of a cluster of Linux machines, each running an instance of the JBoss application server. For each instance, the TAPASCluster JBoss server configuration has been created. The TAPASCluster includes all the new TAPAS services (TAPAS extension middleware in Figure 1) that are intended to extend the standard JBoss application server (the design and the implementation of these TAPAS services are described in more details in "QoS-Aware Application Server: Preliminary Design and Implementation report").

Specifically, in this scenario we will show clients, typically browsers, which use the auction application by issuing HTTP requests. These requests are being sent to a host of the cluster whereby a HTTP load balancer is working as a reverse proxy. Hence, this load balancer is responsible for (i) intercepting all the requests coming from the clients, (ii) choosing the nodes to forward the requests in order to balance the load (dashed lines in Figure 1. This is carried out by the load balancer scheduler which chooses the target node according to an adaptive load balancing strategy), (iii) receiving the response back from the chosen hosts and finally (iv) giving back the response to the client.

Note that, the HTTP load balancer mechanism adopts a primary/backup schema. Hence, the load balancer, implemented as a service of the TAPASCluster configuration, is deployed in every clustered machine, but only one of those services indeed balances the load produced by the clients. The other HTTP load balancer services are backup copies and can be used in case failures occur in the cluster.

The HTTP load balancer is triggered by the Macro Resource Manager of the TAPASCluster server. Hence, we will also show the main Macro Resource Manager's activities that can be summarized as follows.

The Macro Resource Manager will configure the cluster both offline, i.e., after processing a Hosting Service Level Agreement (Hosting SLA) and before the auction application deployment, and at run time. The offline configuration consists in adding new JBoss instances in the cluster in order to meet configuration requirements (for example, in case the initial cluster configuration is not sufficient to meet the hosting SLA), In order to start up new JBoss instances, the Macro Resource Manager will use an auxiliary service, which is not part of the TAPASCluster server. This service is invoked by using the RMI mechanism and deployed in each spare machine, made available for configuration/reconfiguration purposes. In case this service is unavailable (e.g., crashed) in one or more spare machines, the RMI exception mechanism is used to contact another active instance of that service.

The Macro Resource Manager configures the initial cluster (the cluster that will be used for the auction application deployment) by interrogating the Micro Resource Manager so as to get the QoSData related to each node of the cluster. The QoSData are the set of response time, throughput, and JVM free allocated memory parameters. These data are used to produce two important objects that represent the final result of the cluster configuration. These objects are described as follows.

AgreedQoS: the AgreedQoS is the set of response time and throughput parameters that are to be sent to the Micro Resource Manager. Hence, this object represents, for each node of the cluster, the warning point that can be possibly used to trigger a micro reconfiguration.

LbClusterFactor: this object is produced in order to guide, at run time, the load balancer, whose task is to choose the clustered target node to dispatch the client requests To this end, the LbClusterFactor includes, for each node of the cluster, the node name and the corresponding load balancing factor (lbFactor). This latter is a percentage that represents how many client requests each node of the cluster is allowed to satisfy. The lbFactor is computed by considering the aforementioned QoSData, related to each clustered node and obtained by the Micro Resource Manager. Hence, the response time, the throughput and the free memory of each node are combined together so as to produce that load balancing factor.

At run time, we will show how the Macro Resource Manager, together with the Micro Resource Manager, monitors, saves QoS logs and possibly reconfigures the cluster as a response of faults (as failures model we will consider JBoss server crashes only) or overloaded conditions.

In order to demonstrate that the aforementioned activities can provide us with a robust application server, we will compare the TAPASCluster configuration with the standard “all” JBoss configuration. Specifically we will contrast the Apache mod_jk load balancer, installed in front of the standard cluster JBoss server (Figure 2 below) with the TAPASCluster services and the previously described HTTP load balancer. In particular, we will show that the mod_jk module must be statically configured, so as to specify the number of hosts available in the cluster, and its configuration cannot be changed dynamically (i.e., at run time). Hence, if a failure occurs the Apache mod_jk redirects the client HTTP requests, addressed to the crashed node, to another active node in the cluster by using a round-robin policy. As this policy selects a target node with no knowledge of the run time computational load of that node, it is possible that the redirection following a node failure in the cluster leads to overloading another node in that cluster. In principle, this process may continue until all nodes in that cluster are brought to an overloaded state, as a sort of domino effect.

In contrast, we will demonstrate that our TAPASCluster JBoss is robust in the sense that is capable of changing the cluster configuration at run time (i.e., the number of nodes in the cluster can vary dynamically, as a consequence of overload or node failure events, depending on the chosen reconfiguration strategy) and load balancing dynamically the requests depending on the actual load of each available node of the cluster.

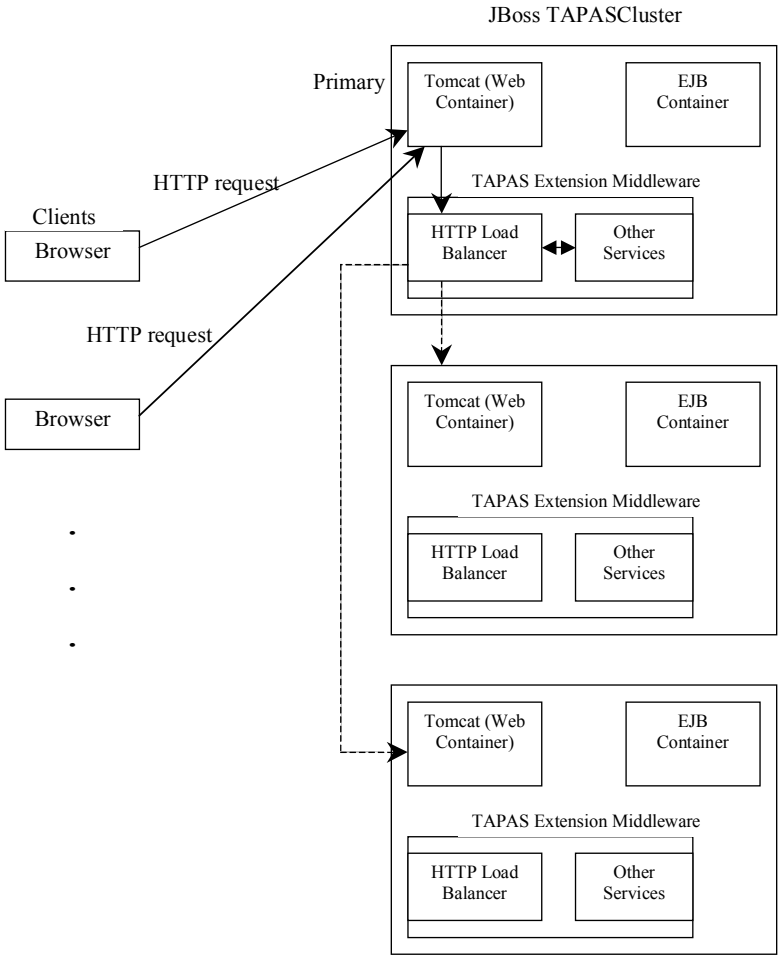


Figure 1: TAPASCluster scenario.

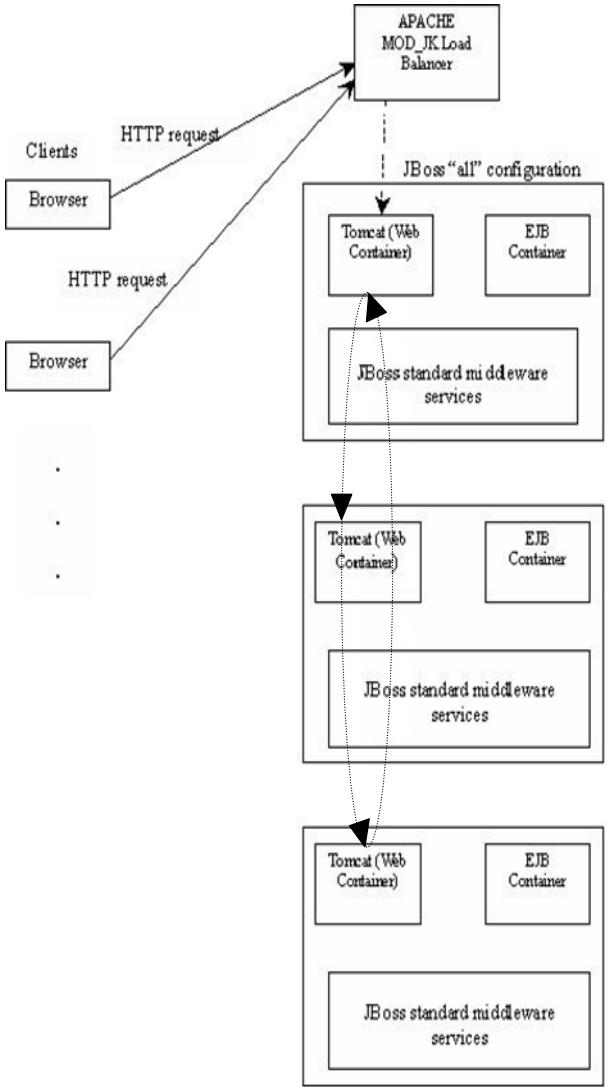


Figure 2: Standard JBoss "all" configuration with Mod_jk.

5. Trusted Coordination: Monitoring Terms and Conditions and evidence generation

Nick Cook and Paul Robinson (Newcastle)

5.1. Introduction

This section describes the demonstration of component-based middleware for trusted coordination. First we provide an overview of the augmentation of the TAPAS demonstrator application with the middleware[1] (full details of the middleware and the component-based JBoss/J2EE implementation are provided in Deliverable D9). Then we describe the processing of a client request by the middleware. The section concludes with an overview of the scenarios that can be used to demonstrate correct operation of the middleware along with a description of the evidence generated.

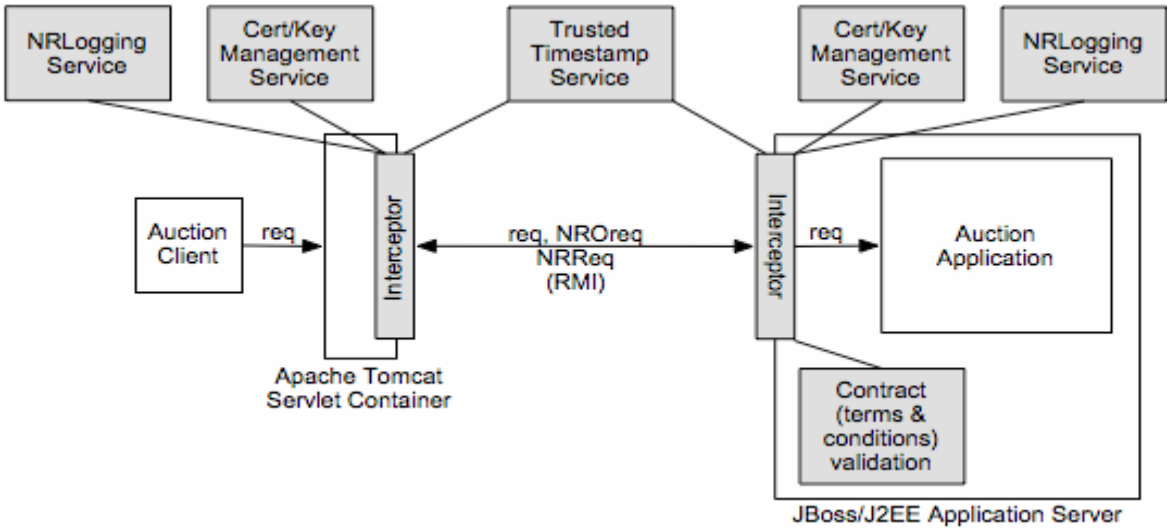


Figure 1. Auction application augmented with middleware for trusted coordination

Auction clients (an auctioneer and two or more bidders) participate in an auction by submitting requests to an instance of the auction application that is hosted by a third-party ASP. In this context, the requirements on the middleware for trusted coordination are that:

1. Both the origin and receipt of requests are rendered non-repudiable. That is, that evidence is generated to irrefutably bind a request to an originating client and that, given evidence of origin, irrefutable evidence of receipt of a request by the auction

[1] By augmentation we mean that the middleware renders the application non-repudiable etc. without modifying the application code

service is generated. The middleware must also maintain (non-repudiation) logs of evidence generated on behalf of client and service.

Submitted requests are validated with respect to business contract terms and conditions that govern the behaviour of clients (auctioneer and bidders) when participating in an auction.

Figure 1 shows the TAPAS demonstrator application augmented with the middleware to meet the above requirements.[2] Client requests are submitted to the auction application through an Apache Tomcat Servlet presentation layer. A client-side interceptor executes in the servlet container. The interceptor is responsible for instantiating middleware to execute protocols for the exchange of non-repudiation evidence. A corresponding server-side interceptor executes at the JBoss/J2EE application server to instantiate the middleware support for service participation in non-repudiation and for request validation with respect to contract. As shown, the middleware accesses services for logging of non-repudiation, certificate and key management, and trusted timestamping (to allow verification of evidence).

5.2. Request Processing

The client-side interceptor traps a client's request for the non-repudiation middleware to process. Evidence of non-repudiation of origin of the request (NROreq) is generated and logged in the client's non-repudiation log. The evidence includes a digital signature over the request data and a trusted timestamp over the signature (to demonstrate that the private key used to sign the request data was valid at time of use). The request and NROreq are then forwarded to the service, along with any certificates required for verification of evidence. The server side interceptor passes the request and NROreq to the non-repudiation middleware for verification and validation. First, the NROreq signatures and related information are verified. If the NROreq passes verification, the evidence is logged. Next, the request is validated against business contract terms and conditions encoded as finite state machines (see Section ?). After validation, a non-repudiable receipt for the request (NRRreq) is generated. The NRRreq includes a signature over the request data and over the result of request validation against contract. The NRRreq is logged in the service-side log and then sent to the client for client-side logging. Finally, if the request was successfully validated, it is passed to the application for processing.

In the auction application, the auctioneer may submit the following requests:

1. Register a bidder — providing bidder identification information and optionally checking credit rating. At least two bidders must be registered for an auction.

2. Create an auction — specifying auction good, auction deadlines, number of bid rounds, opening price etc.

3. Invite bidders to an auction instance (and optionally check credit rating of bidders).

---

[2] For the purposes of this discussion, the application is depicted as executing on a single host and details of clustering etc. are abstracted away.

4. Select an auction winner.
5. Declare an auction failed.

A bidder may:

1. Accept (or reject) an invitation to participate in an auction.
2. Place a bid in an auction round.

For each of the above auctioneer and bidder requests, evidence of non-repudiation of origin and of non-repudiation of receipt (by the application service) will be generated. In addition, the middleware will subject each action to validation with respect to contract terms and conditions

5.3. Demonstration scenarios

In general, when considering the operation of the middleware there are three cases of interest:

1. Normal operation where a participant behaves correctly with respect to contract and generates verifiably correct non-repudiation evidence.
2. Evidence verification failure where, for example, non-repudiation evidence is internally inconsistent or a signature has been generated with a private key that has expired (or been revoked) or is not yet valid.
3. Contract violation where a request is invalid with respect to contract terms and conditions.

Each of these cases can be demonstrated in the TAPAS application as follows:

Normal operation

In normal operation, an auctioneer and at least two bidders will submit valid requests to which the middleware will attach well-formed non-repudiation evidence. The requests will follow the full lifecycle of an auction from bidder registration to auction closure. The evidence for correct operation of the middleware will be that:

1. For each correctly behaving client, there is a full set of entries in both client-side and service-side non-repudiation logs. The logs will in effect be a non-repudiable trace of the execution of the application as represented by the client requests. Each NRRreq entry in a log will include a decision attesting to the validity of the related request.
2. From the point of view of each correctly behaving client, the auction will progress normally to completion, and the application state will change in response to their requests.

Evidence verification failure

To demonstrate evidence verification failure, a bidder will use an invalid key to sign request data. In this case, if the middleware behaves correctly:

1. The bidder's client-side non-repudiation log will contain an NROreq entry but, since the request was mal-formed and failed verification, there will be no corresponding entry in the service-side log.
2. No NRRreq will be returned by the service. Instead, an error will be propagated to the client indicating failure of the request.
3. The request will not be passed to the application and application state will not be changed as a result of the request.

Contract violation

In this case one or more bidders will generate request(s) that violate contract terms and conditions. For example, one auction rule specifies that if an organisation has had a bid accepted in an auction round then it must not bid again in that round. If a bidding organisation is represented by more than one bidder, then a violation of contract occurs if more than one bidder from the organisation attempts to place a bid in the same round. In this case, for all but the first representative from the organisation, the following should result:

1. Both bidder and service-side logs contain NROreq and NRRreq evidence. However, the NRRreq evidence will attest that the related request was invalid with respect to contract.

2. An error will be propagated to the client indicating failure of the request.

3. The request will not be passed to the application and application state will not be changed as a result of the request.

6. Integration of the Group Communication Protocol into JGroups

Antonio Di Ferdinando (Newcastle)

6.1. Architecture of the protocol

The architecture of the protocol that we use is this work is described in detail in [Diferdinando04]. Integration into JGroups, however, required minor changes to the original APIs, mainly concerning terminology and structure. For this reason we refresh protocol's description below.

The protocol is composed out by three separated components that work in tight coordination to offer a reliable service under respect of requested (and agreed) timely guarantees. Division into components is reflected by the general structure, showed in Figure 1, where each component forms a java sub-package whose sum defines the whole package. The three components have been named `Negotiation`, `CoreProtocol` and `Monitoring`. Another sub-package offering auxiliary services has been created, whose description is out of the scope of this document, and again described in full details in [Diferdinando04].

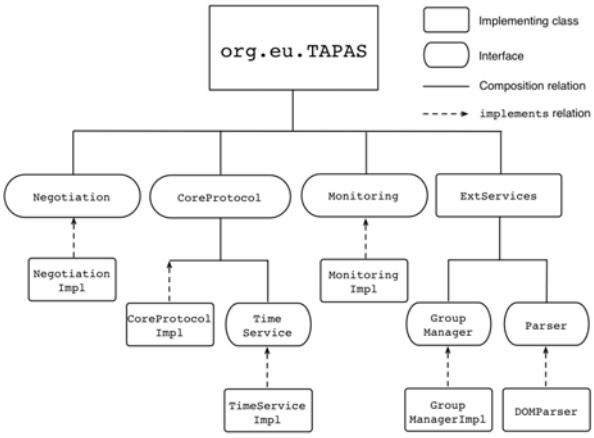


Figure 1: Package structure

APIs for reliable multicast protocol

The algorithm realized offers a QoS-adaptive reliable multicast service. Primitives exported are aimed to multicast messages according to the protocol's specifications, and are then `RMCast(...)` for the sending part and `RMDeliver()` for the receiving part. Real network communication has been defined, as well as all other general objects, by means of interfaces whose implementation make use, in the current release, of datagrams sent over UDP `DatagramSockets`. Characterization of all three sub-services, as well as APIs, are discussed in the rest of this chapter.

Negotiation component

Negotiation component's primary aim is to negotiate a QoS service level with the user and evaluate parameters for `CoreProtocol` component's execution. Its structure implements the `Negotiation` interface by spanning two threads, one providing API for the real negotiation, and another to listen to updates coming from the `Monitoring` component. The first thread, named `Negotiator`, is needed for the real negotiation with the user, and simply calls the `Negotiation` interface to negotiate QoS. This interface's implementation contains all analytical approximation formulas needed to evaluate probability of success based on the user's requested delay. The `Negotiation` interface has then the form shown in Figure 2

```
public interface Negotiation {

    public boolean negotiate(double successProb,
                             double delay,
                         String type);

}
```

Figure 2: Negotiation interface

In this Figure, `succesProb` and `delay` are user's requested probability of success and delay respectively, `type` is the type of delay requested, either `absolute` or `relative`. Please note, with respect to APIs described in [Diferdinando04], the absence of the parameter providing reference to the group manager. The second thread, named `ModelUpdater`, aims to taking track of information coming from the `Monitoring` component and eventually update current probabilistic models for resources upon detection of changes. Update is made by means of the `UpdateModel` interface, which simply calls the layer dependent method update. Figure 3 shows structure of the `Negotiation` component.

Core Protocol component

The Core Protocol component realizes the true QoS-adaptive reliable multicast logic. Two are the main primitives provided by this component: at sending side, the main primitive `RMCast(Message msg, double eta, int rho)` allows the caller to reliably send a message according to the reliable multicast protocol's specifications, while receiving side's main primitive, `RMDeliver()`, allows the caller to reliably deliver a message. Figure 4 shows the structure of the Core Protocol component. Upon instantiation, the `CoreProtocol` spans a thread for the sending part of the reliable multicast protocol (Sender) and a thread for the receiving part (`RMReceiveThread`). These, in turn, call the `RMCast(…)` and the `RMDeliver()` respectively. The former thread waits for the application user to request a message to be reliably multicast (*RMcast*), while the latter listens to a port for incoming messages. Both primitives for reliable sending and receiving a message are described in details in the remainder of this section.

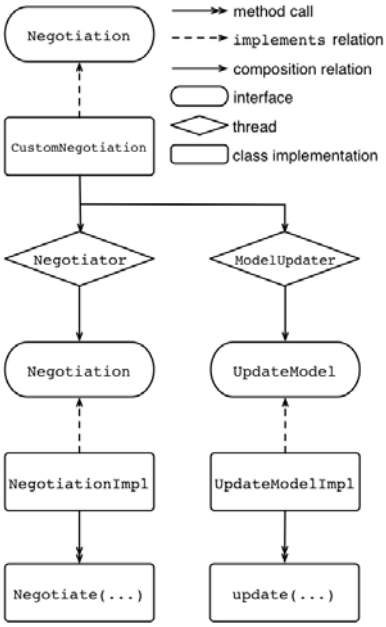


Figure 3: Negotiation component structure

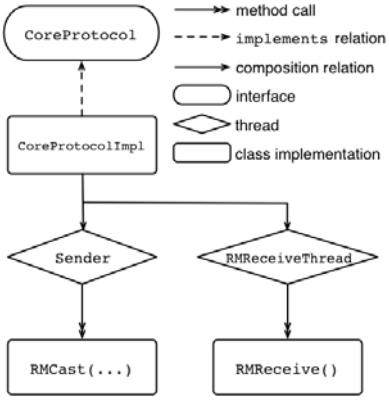


Figure 4: Core Protocol component structure

Reliable Multicast Operation

The Reliable Multicast (*RMcast*) operation is implemented by the `RMCast(Message msg, double eta, int rho)` method. The real message is wrapped into a custom data type named `RMCASTMsg`. Information contained in this data type is shown in Figure 5.

```
public class RMCASTMsg implements java.io.Serializable{
    String originator;  // Message's originator
    String broadcaster; // Message's broadcaster
    int copyNo;         // Level of redundancy
                           realized by this Message
    String msg;         // Real message to send
    long tStamp;        // Timestamp for this message long
    uniqueProcessId;    // Unique ID for this multicast operation
    int sequenceNo;     // Sequence no. for this message
    int rho;            // Level of redundancy
    double eta;         // Gap time for failure independence
}
```

Figure 5: Format for the `RMCASTmsg` data type

The `sequenceNo` field has been included to eventually allow lower level fragmentation of the real message. Level of redundancy, `copyNo`, must be `0<copyNo<r`, while the overall level of redundancy, `rho`, must be `>1`. All other fields are self explanatory.

To send a message according to the reliable multicast protocol, when called, the `RMCast` method invokes a generic Multicaster interface that in this context is implemented by a specific `UDPMulticaster` class. This class is basically composed out by a (synchronized) queue where the message to send is deposed, and a pool of threads that eagerly get the message from such queue and concurrently unicast it to each member. Its structure is shown in figure 6.

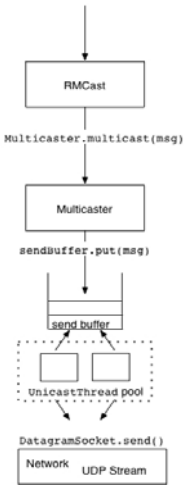


Figure 6: Reliable Multicast operation

Threads inside the pool are given a shared synchronized list of recipients to send the message to. When a thread is in charge of sending a message, gets a copy and extracts the head from the recipients list. It then sends the message to the member referred by the just extracted element, after which it throws the element away and extracts the current head from the list. This process is repeated until the recipients list is empty, after which all threads in the pool die. The purpose of having a pool of threads to send a message instead that a single one is that in this way we can approximate reasonably well a concurrent message sending. According to the protocol's specifications, after some time (η in the protocol description), the process is repeated to send the next copy of the message. The whole process of sending a

message according to the reliable protocol specifications terminates when the original message has been sent ρ + 1 times.

Reliable Delivery Operation

Reliable delivery operation's main primitive is `RMDeliver()`, whose dynamics strictly follows the pseudo code described in [1]. All probabilistic data required for the execution (starting value of *w* for instance) is obtained by the Negotiation component. Reception of a message (figure 7) consists in a `Receiver` object getting the message from the network. Receiver is of course an interface, implemented by the `DatagramReceiver` class that opens a `DatagramSocket` on a port and listens for incoming messages. `DatagramReceiver` puts then the message in a synchronized queue called `ReceiveBuffer`, from which the `RMDeliver` method pops it and treat it according to the reliable protocol specifications. At this level, upon reception of a copy, all support data structures are updated, and the message is stored on a so called *Message Bag* (`MsgBag`). Purpose of this structure is to provide the list of redundant copies received of the same message and is useful in case the originator crashes. The message bag is realized as a linked list of `RMCASTMsgs`. This list is ordered on the copy a message refers to (`copyNo` field of the `RMCASTMsg` data type) and, in case of equality on the broadcaster field, to optimize retrieval. Since the protocol is thought as managing more than a multicast operation at a time, `MsgBags` are themselves stored in a *Message Bag Repository* (`MsgBagRepository`) ordered based on the unique multicast ID. After storing a received message, the process has to set a timeout by which to expect the next copy of a message. This is done by means of a *Time Service*, described below.

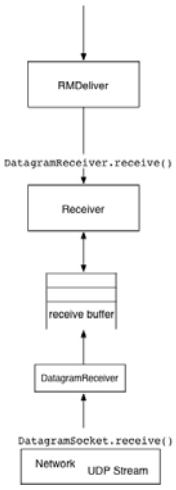


Figure 7: Reliable Delivery of a message

Time Service

The time service is defined by the `TimeService` sub-package, which exports a `TimeService` interface implemented by the `TimeServiceImpl` class, and is locally collocated in the same machine hosting the `CoreProtocol` component. The main reason

for this choice of having it locally rather than remotely collocated is that notification of expiration of a timeout is required to be very timely, since there is a QoS requirement to respect. Besides, local placement of the time service on the same machine hosting the component allows us not to introduce any failure prone link between the component and the time service. In the economy of the protocol, the time service has the sole role of notifying a process that a timeout is expired. This is done by gathering timeout expiration notification requests by means of *Event Listeners*, objects with which a process notifies the time service about its own interest for a determinate event. In the scope of the protocol, the only event a process is interested in is timeout expiration, and interest towards notification of a timeout expiration is notified to the time service by means of a `TimeoutListener` object, shown in Figure 1.8 In this figure, `originalTimeout` is the amount of *millis* to wait for the timeout to expire, `clockedTimeout` is the `originalTimeout` adjusted to the time service's own clock (i.e. the clock of the machine hosting the time service). This is needed in case the owner process' clock is out of alignment with the time service's clock. In this case values of the two fields differ, while in case the two clocks are perfectly aligned, the value is the same. The `serviceTstamp` field contains a timestamp of when the `timeoutListener` has been sent to the time service, and is used when calculating the `clockedTimeout`. All other fields are self explanatory. The `TimeoutListener` class provides, besides methods to set and get values for various fields of the object, a method to notify the owner of expiration of a timeout. When a `TimeoutListener` object is received by the time service, it is inserted by the main thread in a linked list, listeners, containing all `TimeoutListener`s received. Another thread, inside the time service, "ticks" the time by sleeping for a minimum amount of time and, on wake up, matching the current time against the `clockedTimeout` field of each `TimeoutListener` in the list. Timeout expiration is triggered by the current time being equal or bigger than the `clockedTimeout` field of an element. In this case, the time service notifies the owner by calling the `expirationNotification()` method that, on the owner's side, triggers the recovery thread.

```
public class TimeoutListener extends EventListener{
    static long originalTimeout; //timeout to get notification for
    static String owner;         //owner of the listener
    static long clockedTimeout;  //timeout converted into the time
                                 // service clock format
    static long serviceTstamp;   //timestamp assigned by the owner
    static int copyNo;           // copy the timeout refers to
    static long multicastID;     // unique ID of the multicast
    static int ownerPort;        // port of the listener's owner
```

Figure 8: `TimeoutListener` object

QoS Monitoring Component

Reliable Multicast layer is the bottom-most layer in an hypothetical architecture, and at this level the (QoS) `Monitoring` Component is mainly concerned on monitoring the underlying network reliability and performances. Network QoS monitoring is one of today's most challenging task. Heterogeneity of network's nature in terms of traffic and instability (due to protection technologies such as firewalls) make network performance impossible to estimate. All these factors lead to a severe of lack of standards in measurements.

Use of a multicast monitoring tool could be an approach to such a challenging task. For this purpose, a few tools for multicast trees monitoring have been revised. The most promising is the *Multicast Quality Monitor (MQM)* [dressler02], that attempt to monitor QoS across a multicast group by using a combination of pings to measure RTT (Round Trip Time) between nodes and the RTP (Real-time Transport Protocol) [Schulzrinne96] to measure jitter and packet loss (given by RTP). Both these techniques do this by using separate measurement nodes on the group. An alternative to the use of an external tool could be provided by the *IETF IP Performance Metrics*, which tries to develop a standard metrics that can be applied to the quality, performance, and reliability of Internet data delivery services.

6.2. JGroups and integration

Introduction to JGroups

JGroups[Ban99] is a toolkit for reliable multicast communication created by Bela Ban. Its architecture allow the user to join a group by simply using a channel abstraction, used as medium for sending and receiving messages to/from other group members. Figure 9 shows the architecture of JGroups.

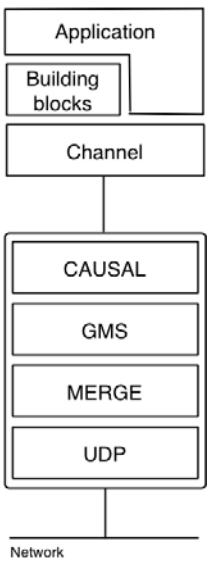


Figure 9: Architecture of JGroups

The Figure shows that JGroups is composed out of three main components: a *Channel API* used by programmers to build group communication applications, *Building Blocks*, a set of more sophisticated APIs realized on top of the channel to abstract the (simple) primitives represented by channels, and a *protocol stack* that realizes the true requested service by means of protocol composition. From our context point of view, integration process has involved only changes in the latter component, i.e. the protocol stack, so we will only focus on it redirecting the reader to [Ban99] for any further detailed description of the first two components.

To be operative a channel must be instantiated, after which it can be connected to a group. Upon creation a channel spans the protocol stack, to be used throughout the channel's life-cycle, composed out by several protocols that work in tied coordination to achieve the communication purpose. The protocol stack is defined by means of a string (the *definition string*) the user specifies, containing a list of protocols the user wants to use, together with each protocol's required parameters. The definition string is then passed as parameter to the channel object in creation phase. Figure 10 shows the string used to span the default protocol stack, i.e. the protocol stack used when the user does not specify any custom one. Each protocol specified in this string takes care of a determinate task in the group communication process.

```
String props="UDP(mcast_addr=228.8.8.8; mcast_port=45566;ip_ttl=32;"+
             "mcast_send_buf_size=64000;mcast_recv_buf_size=64000):"+
             "PING(timeout=2000;num_initial_members=3):"+
             "MERGE2(min_interval=5000;max_interval=10000):"+
             "FD_SOCK:"+
             "VERIFY_SUSPECT(timeout=1500):"+
             "pbcast.NAKACK(max_xmit_size=8096; gc_lag=50
                retransmit_timeout=600,1200,2400,4800):"+
             "UNICAST(timeout=600,1200,2400,4800):"+
             "pbcast.STABLE(desired_avg_gossip=20000):"+
             "FRAG(frag size=8096;down thread=false;up thread=false):"+
             "pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;"+
             "shun=false;print_local_addr=true)";
```

Figure 10: Definition string for the default protocol stack

The protocol stack is defined starting from the bottom: the first protocol defined (`UDP` in figure 9) is the bottom-most protocol in the stack. Subsequent protocols, then, occupy a

crescent higher position in the stack. The last protocol specified in the string is the higher-level one and, in this example, is a group management protocol (`GMS`). Description of the whole suite of protocols is outside the scope of this document, so we will only describe protocols of interest for the scope of this document, redirecting the reader to [Ban99] for full details

UDP in JGroups

The UDP protocol object in JGroups realizes a connectionless transport layer based on datagrams. In the definition string the user is given two options of using connectionless datagram communication.

In the first option, the user can choose to use IP multicast[NWG86] communication (as in figure 10), with which a message is sent to an IP multicast address. Such message is then automatically multiplexed in as many copies as members of the group (i.e. the IP multicast address) are and delivered. Reliability is not a native concept in IP multicast, and it but be provided with some techniques. JGroups uses a NAK-based system, where missed reception of a packet is denoted by sending a Negative ACK (NACK) to the broadcaster, with which it is requested to unicast the requester with the missing copy. In the actual implementation, this recovery mechanism is subject to the use of two other protocol objects. `NAKACK` is concerned to NAK-based retransmission of missing messages, and actual retransmission of missing messages is subject to the use of `UNICAST`, concerned to unicast communications. Absence of one or both of the aforementioned protocols implies inability to operate the respective task, and the correct use of IP multicast is then tied to the use of all three protocols. Technically, communication is done by mean of a `MulticastSocket` that eases the user from the task of message multiplexing.

In the second option, the user can choose to use traditional datagram based communication, coordinated by a *Gossip Server* centrally handling the groups. A gossip server is a central entity handling group management. It contains the current view of the group and keeps consistency of each member's view by broadcasting the current view upon changes. Each group member constantly registers to the Gossip Server as to notify his aliveness. Missing registration triggers the gossip server to broadcast a suspicion message. Multicast communication is realized by means of concurrent unicast transmissions on a `DatagramSocket`. It's easy to understand the key role of the Gossip Server to keep each member with consistent views of the group.

In the context of our work, the nature of the algorithm used by our protocol makes if perfect for the use on a UDP-based system without use of IP multicast. The model our protocol applies to, in fact, make it unsuitable for the use with IP multicast since identity of each party of the group is key for the recovery mechanism.

Sending and receiving datagrams in JGroups

When the user wants to multicast a message, it uses the `send(Object obj)` method of the `JChannel` class that sends it through the protocol stack, starting from the topmost protocol. The message is then propagated through the stack. Each protocol that needs to

process the message appends a header to witness occurred processing. The message then reaches the `UDP` layer and is then sent over the network.

All types of information are propagated throughout the whole stack as an `Event`. In JGroups an event is a composite object containing a `type` and an `argument` objects. The type defines a category the argument refers to, while the argument encapsulates the real information. Type `Event.MSG` defines information coming from the user application. Usually this event needs only to be multicast so each protocol of the stack propagates it to the lower protocol[3]. When reaching the `UDP` layer, the argument is extracted from the `Event` and encapsulated into a lower level `Message` data type, together with information about source and destination and, after insertion of `UDP` header, sent.

The reverse process is adopted at receiving side. Once received at `UDP` layer, information is extracted (as well as `UDP` header), encapsulated into argument of an `Event.MSG` type and delivered to the upper layer.

A system so built is highly dynamic and flexible. Reliability and completeness is achieved by means of a fairly cumbersome protocol stack, causing a general slowness making so the system unfeasible for the use in contexts where timeliness is required and overall must be guaranteed. Besides, the slowness is accentuated if the system is used in a WAN environment, typically subject to heavy and unpredictable fluctuations.

6.3. Integration

As mentioned in earlier sections, in our context the *protocol stack* holds the position as the most important component. Technically it is composed out by a series of objects each one extending the `Protocol` class, which realises the common inter-protocol logic.

From the structural point of view, integration means addition of another protocol to the actual protocol suit, which involves a change in the way the definition string must be defined. From the point of view of the dynamics of JGroups, integration changes the role of the `UDP` protocol. If such protocol, in fact, was previously only concerned with networking, now it is extended with some logic by means of which it *decides*, based on both user request and protocol stack properties, the use of the QoS-Adaptive reliable multicast protocol as we shall describe shortly.

We proceeded in two main phases: in the first phase we created a brand new protocol[4] named `RMCAST` containing the sending and receiving logic of our protocol, while in the second phase we conbined `RMCAST` with the rest of the protocol suite. The remainder of this section describes both phases in more detail.

---

[3] Unless the message needs to be retransmitted upon intervention of the `NAKACK` protocol, or is subject to intervention of other protocols.

[4] From now on we will use the word protocol to intend a JGroups layer, unless stated differently.

Phase one: the RMCAST protocol

The new `RMCAST` protocol is meant to be a transport layer alternative to both IP multicast and traditional UDP , to be used in situations where timely reliability is required. Figure 11 show the new default protocol stack, after integration of `RMCAST`.

Figure 11: Protocol stack with `RMCAST`

It can be instantiated only together with `UDP` and without the use of IP multicast, for the reasons mentioned in previous sections. If there is a mismatch in the protocol stack definition string (either the user uses `TCP` rather than `UDP` or uses IP multicast), `RMCAST` becomes an empty transparent object. Figure shows the new definition string.

The first thing to note, along with the presence of `RMCAST` explained below, is `UDP`'s declaration not to use IP multicast, together with the different use of `PING`[5]. If in Figure 10 it was just a mean to declare service timeouts, now it becomes the way to localize the Gossip Server. Parameters of `RMCAST` contain user-requested QoS level (delay amount and type, as well as success probability). All other data needed by the protocol (mainly network related) initially taken from an XML file, and eventually updated by the `Monitoring` component. Upon creation, the input data is acquired and processed, and the negotiation process starts. If unsuccessful, the protocol is not instantiated (i.e. the whole object is *null*).

The user is notified about failure of negotiation, and UDP will be aware of the unsuccessful negotiation so as to behave as RMCAST does not exist. If successful, negotiation will finish with proper values for the redundancy level (r) and subsequent retransmissions gap time (h). Threads to manage sending and receiving will be instantiated, after which the protocol will be ready for use.

[5] This protocol is used to discover initial membership references.

```
String props="RMCAST(success_prob=0.99;user_delay=300;
               delay_type=absolute;rmcast_buf_size=64000):"+
             "UDP(ip_mcast=false;mcast_addr=228.8.8.8;
               mcast_port=45566;ip_ttl=32;"+
             "mcast_send_buf_size=64000;mcast_recv_buf_size=64000):"+
             "PING(gossip_host=localhost;gossip_port=5555;
               gossip_refresh=1000;"+
             "timeout=2000;num_initial_members=2):"+
             "MERGE2(min_interval=5000;max_interval=10000):"+
             "FD_SOCK:"+
             "VERIFY_SUSPECT(timeout=1500):"+
             "pbcast.NAKACK(max_xmit_size=8096; gc_lag=50
               retransmit_timeout=600,1200,2400,4800):"+
             "pbcast.STABLE(desired_avg_gossip=20000):"+
             "FRAG(frag_size=8096;down_thread=false;
               up_thread=false):"+
             "pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;"+
             "shun=false;print_local_addr=true)";
```

Figure 12: New default definition string, with `RMCAST` integrated

When initialized, `RMCAST` instantiates objects to build its own architecture, described earlier in this document, and waits for a request of an object to be `RMcast`. At the lowest level of the sending side, the `UDPMulticaster` object creates and initializes a (datagram) socket to send datagrams. Such a socket is completely different from the one used by UDP and used only to `RMcast` messages. The socket created at UDP for unicast communication (used by the `UcastThread` object, proposed to handle unicast bits of the communication process) is still used to exchange all information not intended to be `RMcast`.

Figure 13 shows, with respect of the bottom part, differences between the original default protocol stack and the same default protocol stack after `RMCAST`. The user specifies the wish to use `RMCAST` by setting a boolean field (namely `RMCast`), in the Message object created for the operation, before entering the protocol stack. At UDP, in fact, there is no distinction between a Message received from a `UcastReceiver` and one received from the upper protocol and a check on the sole `RMcast` field would result in the same Message continuously retransmitted. If this field is set to false the object is treated as if `RMCAST`

would not exist. Upon reception of an event from UDP, a check is performed. If the message comes from the upper protocol and has the boolean field set, the message is from the user and has to be `RMcast`. In this case, UDP simply propagates the message down in the stack, to `RMCAST`, where the Sender object will multicast it according to protocol's specifications using its own socket. In order to facilitate things at recipient side, another Boolean field named `already_rmcast` is set to determine whether the Message has already been RMcast or not. A message coming from the upper protocol but with `RMCast` not set denotes a service message and is then handled traditionally, i.e. sent by UDP. If passed down to `RMCAST`, the message is extracted from the `Event`, converted to Message and `RMcast` according to the protocol specifications.

When received, the `Message` object is extracted from the datagram packet. Once inspected and detected the type of information it contains, a decision on how to treat the `Event` encapsulating the `Message` is made. Again, if `RMCast` is not set the `Event` is treated traditionally and it is propagated up in the stack. Otherwise, if the same field is set, along with `already_rmcast`, there is no doubt that the message is a data message RMcast from another group member, in which case is propagated down to `RMCAST`. Here it will be handled by the `RMReceiverThread` object which realizes the receiving logic of the protocol, described in the first part of this document.

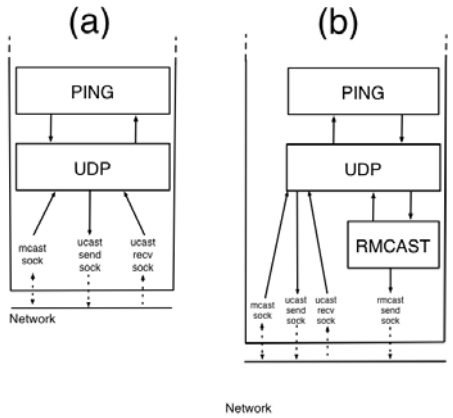


Figure 13: Bottommost part of the protocol stack, before (a) and after (b) integration

Phase two: putting it all together

Integration of `RMCAST` in JGroups required few changes in the original APIs described in [Diferdinando04]. Main ones concerned group management and low level objects format and, are discussed in the next paragraphs.

Group management

Integration into JGroups implied the use of JGroups own group management protocol rather than the one described in [Diferdinando04]. Amongst other things, this implied the use of JGroups' `Address` objects in place of the "old" `Member` objects. An `Address` object is interface that abstracts and generalizes the concept of address. Currently JGroups realizes the concept of address by means of an `IpAddress` object, which binds the definition of address

with an abstraction based on IP addresses. Figure 14 shows its structure. The fundamental fields in this object are `ip_addr` and `port`, which are self explanatory and uniquely define a group member's address, while `additional_data` and `sAddrCache` fields give the possibility to include, the this object, extra information and history respectively.

```
public class IpAddress implements Address {
    private InetAddress ip_addr=null;
    private int port=0;
    private byte[] additional_data=null;
   protected static HashMap sAddrCache=new HashMap();

    [...]
}
```

Figure 14: Structure of an `IpAddress` object

Conversion from `Member` to `IpAddress` involved a change in the way to modify and get information from the object. All methods previously dealing with Members have been conformed to the use of `IpAddresses`. `A` data structure containing information on other members of the group has been created and the way of obtaining group views has changed. If the original use of the `GroupManager`, in fact, foresaw the client asking the RMI registry for a group view, JGroups realizes a mechanism by which the Gossip Server sends events notifying the change of the group view. To conform to this mechanics, `RMCAST` had to be extended as to handle view change events too, in such a way to have an up-to-date view of the group handled consistently with the rest of the protocol suite. This structure has the form of a `Vector`. When `UDP` receives a message of type `Event.VIEW_CHANGE`[6] it upgrades its view and forwards it to `RMCAST` that, in the same way, upgrades its view (realized by the aforementioned `Vector`) and deletes it.

Low level objects format

The original version of the QoS-adaptive reliable multicast protocol encapsulated information to RMcast in a custom type named `RMCASTmsg`, whose structure contained fundamental information for the protocol such as originator, broadcaster, both level of redundancy realized by that message and overall, time to wait for next retransmission etc[7]. Obviously JGroups already contained a custom low level type, named `Message`, used to exchange information between group members. This structure contained information about

[6] This event type encapsulates a message coming from the Gossip Server and is used to guarantee consistency in the group view of each member.

[7] Consult [Diferdinando04] for full description of the object.

source and destination of the Message, body of the message, a series of headers, an overhead due to addresses and a serial version. src_addr=null indicates that the message has to be multicast. As should be well known by now, our protocol needs a different set of information for its correct execution. Rather than using two separate low level data types, we decided to join them together: we extended JGroups' Message data type as to include information needed by our protocol. Structure of the extended Message object is showed in Figure 15.

```
public class Message implements Externalizable {
    protected Address dest_addr=null;
    protected Address src_addr=null;
    protected byte[] buf=null;
    protected HashMap headers=null;
    static final long ADDRESS_OVERHEAD=200;
    static final long serialVersionUID=-113736403583284703 4L;

    //RMCAST-specific data
    protected boolean rmcast = false;
    protected String originator=null;
    protected String broadcaster=null;
    protected int copyNo=-1;
    protected long tstamp=-1;
    protected long uniqueProcessId=-1;
    protected int rho=-1;
    protected double eta=-1;

    [...]
}
```

Figure 15: Structure of the extended JGroups Message type

Important things to note, here, are the absence of the SequenceNo, originally included in the protocol's Message data type, and the contemporary presence of src_addr, originator and broadcaster. SequenceNo has not been included because JGroups uses a higher level fragmentation protocol (namely FRAG) which already provides frag/defrag capabilities at another level, making the field unnecessary. The presence of the src_addr, originator and broadcaster fields has been decided to keep clearly separated the general from the RMCAST-specific part of the object. if a Message is not

intended to be RMcast it will not be forwarded to RMCAST and the RMCAST-specific fields will remain set to null values and the object will work exactly like it was working originally.

On creation phase all RMCAST-specific fields set to null values and the object works exactly as before integration. These are then filled once the Message enters RMCAST's own architecture, just before being sent, and act as a reference for information about RMcast conditions.

6.4. Benefits to Jgroups

Benefits of integration of our protocol into JGroups can theoretically be significant. First, since the protocol offers a true QoS-adaptive reliable multicast communication, it could make JGroups suitable for the use in WAN environments without the need of keeping connections with other members alive (i.e. without the use of TCP). From a more structural point of view, our protocol enhances JGroups' bottom most stack layer. Use of RMCAST, in fact, alone guarantees delivery of the message to all (or none) of the operative processes, letting the use of the STABLE[8] protocol become unnecessary. Stability in fact lies in the guarantee that a message has been delivered by all group members, and RMCAST provides such guarantees as a native feature in its algorithm. Besides, the possibility to require a probability of success as close to 1 as wanted, limits the possibility that a message is lost amongst a sequence of messages. As a consequence, the use of NAKACK might become negligible and, as well as the use of UNICAST. Finally, the model our protocol applies to do not require to know if a member becomes faulty during the operation. In fact, the nature of the protocol guarantees that the multicast operation will terminate even if there are as many partitions as group members in the group, limiting possibilities of unsuccessful protocol termination to unpredictable conditions (hardware faults etc.). Given this, the use of our protocol would probably not require the use of FD_SOCK[9] and VERIFY_SUSPECT[10] protocols, since for our protocol is not fundamental to know if a member becomes inoperative.

References

[Diferdinando04] Di Ferdinando A., Ezhilchelvan P.D., Mitrani I. and Morgan G., "QoSadaptive Group Communication". TAPAS Deliverable D8, Brussels, May 2004. www.newcastle.research.ec.org/tapas

[dressler02] Dressler F. MQM - Multicast Quality Monitor. In Proceedings of 10th International Conference on Telecommunication Systems, Modeling and Analysis (ICTSM10), vol. 2, Monterey, CA, USA, October 2002, pp. 671-678.

[8] This protocol is used to determine if a message is stable or not.

[9] This protocol realized a failure detector at socket level.

[10] This protocol is concerned of verifying that a suspected member is really faulty, and works tightly with the FD_SOCK protocol.

[Schulzrinne96] .Schulzrinne H. et al. RTP: A Transport Protocol for Real-Time Applications, RFC 1889, IETF, January 1996.

[Ban99] Ban B., "JavaGroups User's Guide". http://www.javagroups.com

[NWG86] Network Working Group, "Host Extensions for IP Multicasting" RFC 988, July 1986. http://www.faqs.org/rfcs/rfc988.html

7. QoS Enabled Real-Time Collision Detection

Graham Morgan, Kier Story (Newcastle)

7.1. Introduction

We present a real-time collision detection service that supports QoS guarantees suitable for satisfying the collision detection requirements of graphically represented 3D virtual worlds. Our approach is a distributed one, allowing our real-time collision detection service to scale to support complex virtual worlds via the use of clustered servers. We base our service on QoS enabled group communications middleware to enable a deterministic approach to satisfying the QoS requirements of virtual worlds. Our approach is adaptive in that we always guarantee real-time requirements of a virtual world via the adaptation of the accuracy of collision detection in runtime to offset varying processing availability or message latency.

7.2. Collision Detection

In this section we describe the basic principles of the collision detection algorithm used in our approach and how our approach may be tailored to satisfy the collision detection requirements of a virtual world. We assume a broad/narrow phase approach to collision detection. The broad phase is used to discount pairwise comparisons between objects that are separated by a distance (determined by developer) that indicates that collision has not occurred. The broad phase uses non-precise means of collision detection in that individual parts of objects are not compared, making this type of collision detection low cost compared to the narrow phase. The narrow phase considers only those pairwise comparisons identified by the broad phase. Narrow phase collision detection attempts to determine point of contact between object pairs (possibly at the polygon level).

Broad Phase

The purpose of the broad phase is to eliminate as many pairwise comparisons as possible so the narrow phase does the minimum of wasteful collision detections (objects incorrectly determined to have collided by broad phase). We base both our narrow and broad phase collision detection on binary space partitioned trees (BSP trees). BSP trees provide a convenient structure to store polygons that make up a scene (view of a virtual world) and may also aid the process of collision detection. BSP trees may be used to reduce the rendering time of a scene by ensuring only those polygons that are visible are drawn. In collision detection, BSP trees can be used to determine the proximity of objects within a virtual world and so aid in the broad phase collision detection test. Furthermore, BSP trees may be used in the narrow phase collision detection test by comparing BSP trees that represent two objects.

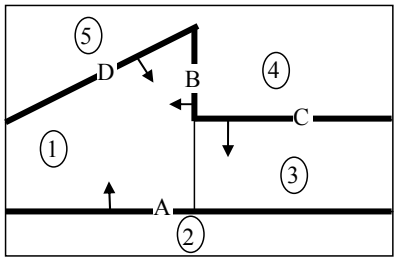


Figure 1 – Dividing the virtual world with lines.

We now provide a brief explanation of BSP trees, emphasizing the benefit they have for broad phase collision detection. Figure 1 highlights a 2D environment that we shall use to illustrate our example (BSP trees may be used in 3D environments). Lines are used to dissect the virtual world (in 3D we use planes). Each line partitions the virtual world into halves. A halve may then be partitioned again, providing ever increasing sub-division of the virtual world. We subdivide the virtual world in this manner in a predefined order. The lines used to divide the virtual world in our example have been applied in alphabetical order. We determine if lines fall to the "left" or the "right" of lines previously used to divide the virtual world. In our example this provides the binary tree shown in figure 2.

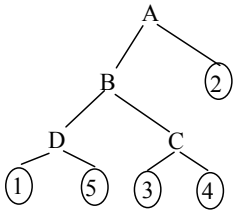


Figure 2 – BSP tree derived from diagram in figure 1.

From figure 2 we may determine which partition an object may lie: objects 1 through 5 have been placed in their appropriate leaf nodes of the tree. By applying this approach as the broad phase of our collision detection technique we can identify which objects are close to each other (share the same leaf node or neighbouring leaf nodes). Leaf nodes identify the sets of objects that the narrow phase of the collision detection algorithm must consider for pairwise comparisons.

Narrow Phase

The narrow phase of the collision detection applies a similar technique as the broad phase. However, we do not consider objects in their entirety as in the broad phase, but create the BSP tree that reflects the relationships between the polygons that contribute to the construction of a 3D object.

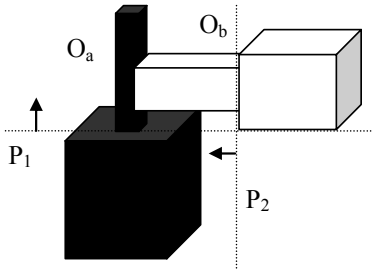


Figure 3 – Colliding objects.

Figure 3 shows the collision between two 3D objects (O_a and O_b). It is clear that one end of O_b is colliding with the top part of O_a. If we had not entered the narrow phase and just acknowledged the collision identified by the broad phase it would not be possible to identify the point of contact between O_a and O_b. Identifying the point of contact is necessary in many simulations as the collision response (assuming objects cannot share the same space and so must react to collisions) may be different depending on where objects appear to have collided.

In figure 3 we shall use a simple example to highlight the advantage of BSP trees for determining point of contact. For clarity we shall not show the complete BSP trees for the objects, but will concentrate on two plains of division (P_1 and P_2) to aid in identifying approximate point of contact. Objects O_a and O_b are each constructed from two distinct shapes. By placing dividing planes at the join of these two shapes for each object we may determine that it is the long thin rectangles that belong to each shape that are colliding and that collision is occurring above O_a and to the left of O_b (as these will be co-located in the same leaf node). The addition of planes would be needed for more accurate collision detection.

Quality of Service Enabled

Real-time collision detection is a time consuming task that requires substantial processing resources to support complex virtual worlds. Such worlds may contain thousands of objects. The objects themselves may contain thousands of polygons to represent realistic representations of everyday entities. For example, a human figure is commonly rendered using over one thousand polygons in modern computer games to attain a sense of realism. Real-time collision detection must be achieved within strict time limits so as to avoid objects sharing a degree of virtual space that renders the simulation of a virtual world unrealistic (e.g., fingers passing through walls). Therefore, a desirable property is to allow a developer to instruct a collision detection service to complete within a given time. An approach to ensuring a collision detection service satisfies timely requirements is to reduce the accuracy of collision detection. As the broad phase deviates little with respect to time (placing objects in BSP tree of static size) and must consider all objects (missed collisions must never occur), then it is the narrow phase that is reduced in accuracy.

Reducing the accuracy of the narrow phase in our approach is based on varying depth traversals of the BSP tree associated to object polygons: the lower down the tree we traverse

the more accurately we can determine exactly which polygons are colliding between objects, fewer levels of the tree traversed results in a more general determination of point of contact. This is clear in figure 3 where we have identified, via P_1 and P_2, that collision is occurring somewhere between the two small parts of the objects O_a and O_b, but not the exact point of contact. Identifying the exact point of contact would require a deeper traversal of the BSP tree. During runtime, based on processing resources available, the complexity of the virtual world objects and the number of objects the broad phase has passed to the narrow phase, a judgement on how deep BSP tree traversal should be in the narrow phase can be made.

7.3. A Distributed Approach

Deploying a collision detection service on a single node provides for straightforward assessment of how deep tree traversal should be in the narrow phase. This is because all parameters required to assess the processing resources are readily available and vary to a degree that is insignificant during the execution of a virtual world simulation (e.g., processing resource, rendering time). However, complex virtual worlds that support many thousands of highly detailed objects need a high end server to satisfy processing resources. Therefore, a desirable approach would be to utilise server clusters in the same manner as e-commerce Internet sites (e.g., Google™) to provide low cost scalable computing. Unfortunately, such clusters are constructed on LANs that exhibit varying degrees of latency over time. Such latency is much higher than found within a single multi-processor machine (commonly used to support complex virtual worlds), and so has to be accounted for when determining the QoS of the real-time collision detection.

System Architecture

A real-time collision detection service that is to be deployed over a standard cluster requires a group communication sub-system that has the ability to support pre-defined guarantees of Quality of Service. Furthermore, the QoS of such a service should be adaptable during real-time to reflect the varying QoS that is present of the underlying LAN, informing our real-time collision detection service to alter the depth of narrow phase tree traversal to satisfy timely requirements of the virtual world as and when necessary. The QoS enabled group communication service that supports all these qualities and provides the basis on which we construct our collision detection service has been developed in Newcastle University as part of the EU TAPAS project.

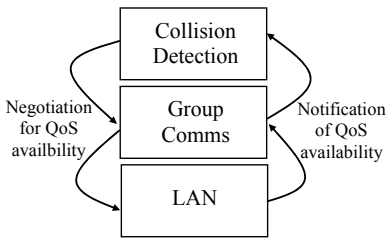


Figure 4 – Abstract view of protocol stack.

Figure 4 shows how the individual components interact and form a protocol stack for the prorogation of messages throughout the cluster. Initially the collision detection service requests QoS guarantees from the group communication service. This will be in the form of upper bound message latency and reliability of messages. In turn the group communication service determines if such a request is possible by examining the performance of the LAN (derived from historical data collected by the group communication service or supplied by the LAN itself). Notification of QoS availability is supplied up through the stack. Based on what QoS guaranteed are offered by the group communication service, the collision detection service determines what depth tree traversal should be associated to the narrow phase of collision detection. The negotiation/notification cycle may be repeated during run-time if the collision detection service or the LAN indicates a change in resources (e.g., virtual world becomes more complex or LAN becomes congested).

Deployment Architecture

Figure 5 presents a diagram outlining the description of our architecture. We assume a number of nodes co-located in a single LAN will satisfy the narrow phase of the collision detection requirements. The virtual world is spatially sub-divided into regions with each node in the collision detection cluster responsible for determining narrow phase collision detection tests for a set of objects contained in a single region of the virtual world. This spatial sub-division represents the broad phase of our collision detection approach. A region is unique to a node and only a single node will attempt narrow phase collision detection tests for the same two objects during an iteration of narrow phase collision detection (between one frame of animation and the next). An object may appear in multiple regions (the case if an object overlaps region boundaries). This may result in an object pair appearing in more than one region. If this is the case then only one node will enact the narrow phase collision test. This eliminates the possibility of duplicate narrow phase collision tests carried out on distinct nodes.

A single node exists (collision detection server) that assumes responsibility for identifying which region(s) an object is associated with and informing the appropriate collision detection node(s) of the objects that they must consider for narrow phase collision detection. The collision detection server carries out the broad phase collision detection test.

We assume a set of nodes that support the virtual world and allow users to interact with the virtual world (user domain). Via a node, each user may control one or more avatars that may interact with objects that populate the virtual world. We classify objects into the three following categories:

- **Avatars** – objects that move under the direct control of a user.
- **Active** – objects that move in some pre-determined manner as directed by some programmed logic associated with the virtual world.
- **Non-active** – objects that do not normally move but may move due to collision with another object.

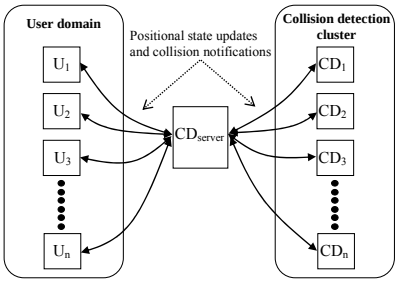


Figure 5 – Collision detection via cluster.

Given our assumptions on the types of objects that may exist in a virtual world we support such objects in a distributed manner in the following way:

- Nodes in the user domain are only responsible for location updates and the collision response of their avatars with active and non-active objects equally distributed across nodes in the collision detection cluster. A node in the collision detection cluster is responsible for location updates and collision response management of their local active and non-active objects.

All active and non-active objects are distributed across nodes in the collision detection domain and responsibility for avatars is distributed across nodes in the user domain. Each node in the collision detection domain is responsible for the location updates and collision responses of a subset of active and non-active objects located in the virtual world. This approach requires that the application state level update messages associated to active and non-active objects be propagated to the appropriate user nodes from the collision detection nodes. The collision detection server receives all application state update messages sent from the collision detection domain and relays these messages to the appropriate recipient nodes in the user domain.

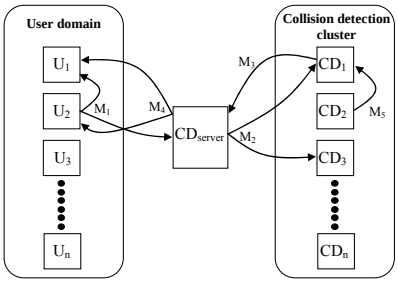


Figure 6 – Message dissemination.

Figure 6 describes an example of message passing that may occur. We assume an optimization in that nodes in the collision detection domain may inform each other of

active/non-active object traversals of spatial division boundaries associated to collision detection. For example, in figure 6 CD_2 notices that an object, say obj_1, it hosts has traversed a spatial division boundary that indicates that obj_1 and associated collision detection should now hosted by another collision detection node. Therefore, CD_2 multicasts a message (M_5) that informs all other nodes that obj_1 has now left its spatial subdivision and indicates the latest position in the virtual world. There is no need for CD_2 to realise what other node is responsible for the spatial subdivision obj_1 has moved into. This is a flexible approach as nodes may be added and removed during runtime to augment the processing capabilities of the system without having to announce reconfiguration of the spatial subdivisions to all nodes. For example, assume there are a disproportionate number of objects compared to the rest of the virtual world within the subdivision managed by CD_2. A new collision detection node, say CD_7 may be introduced that divides the spatial subdivision currently managed by CD_2 (resulting in one leaf node been divided into two leaf nodes (a deepening of the broad phase tree). This approach to object hosting ensures that collision detection nodes only need to inform each other when objects traverse spatial boundaries and not when objects move as would be the case if a collision detection node assumed hosting responsibilities for an object throughout the lifetime of the virtual world.

In figure 6 the collision detection server participates in application level state update message passing as a receiver only. The collision detection server consumes all messages associated to positional update information of all objects associated to all user nodes. As positional state update messages are received (M_1) the collision detection server sends (using QoS enabled group communications) the positional update message (M_2) to appropriate CD nodes (in the example only CD_1 and CD_3 are interested in the message). If any collisions have been identified by a CD node then a message is sent to the collision detection server indicating such collisions (CD_3 sending M_3). The collision detection server then issues a message to the nodes where collided objects have been identified (M_4).