State Transmission Mechanisms for a Collaborative Virtual Environment Middleware Platform

João Orvalho[1], Pedro Ferreira[2] and Fernando Boavida[3]

Communications and Telematic Group
CISUC – Centre for Informatics and Systems of the University of Coimbra
Polo II, 3030 COIMBRA – PORTUGAL
Tel.: +351-239-790000, Fax: +351-239-701266
E-mail: {orvalho, pmferr, boavida@dei.uc.pt}

Abstract. Collaborative virtual environments (CVE) require the use of specially designed mechanisms that allow for a consistent sharing of state among involved users. These mechanisms must, somehow, compensate for network latency and losses in such a way that all players have a single, coherent perception of the system state. Common middleware platforms have difficulty in guaranteeing this consistency, and this is the prime reason why the main research topic for CVEs is the efficient, scalable and reliable transmission of state information. This paper presents a state transmission framework developed for a middleware platform that was constructed by the authors in an earlier project [1]. This middleware platform, Augmented Reliable Multicast CORBA Event Service (ARMS), which already supported several QoS adaptation mechanisms and reliable transmission in multicast environments, was extended with CVE-oriented state transmission mechanisms. After an identification of key requirements of collaborative virtual environments, the relevant features of the proposed state transmission framework are presented. This framework has been integrated in the ARMS platform and was subject to a series of performance tests whose results are included and discussed in this paper. The paper ends with a summary of contributions and an identification of guidelines for further work

1. Introduction

One fundamental problem of collaborative virtual environments is the maintenance of a consistent shared state over interactive distributed media. The current state-of-the-art approach to achieve consistency in CVEs is to use dead reckoning [2, 3]. Nevertheless, in [3] it is demonstrated that traditional dead reckoning mechanisms may fail in ways that cause significant harm to the overall state of CVEs. Network latency and losses contribute to the difference between the predicted and the real system state, and this difference may significantly exceed the threshold that triggers

[1] College of Education, Polytechnic Institute of Coimbra
[2] Polytechnic Institute of Tomar
[3] Informatics Engineering Department of the University of Coimbra

state transmission, resulting in state inconsistencies. One of the main research topics for CVEs, from a quality of service (QoS) point of view, is how to efficiently transmit update messages so as to provide scalability, minimized delay, consistency and reliability [4].

Mauve proposes the concept of *local lag* [5] to prevent inconsistencies, and the *timewarp* algorithm [3] to keep state consistency, possibly in combination with dead reckoning. The *local lag* is a simple mechanism: "instead of immediately executing an operation issued by a local user, the operation is delayed for a certain time before it is executed" [5]. The fixed amount of local lag constitutes a drawback of this method. In the *timewarp* method [6] each participant saves the state at certain moments in time. When an inconsistency occurs, the state is rolled back to the state that immediately preceded the operation that caused the inconsistency. After that, the medium is played (fast) forward until the current medium time is reached. This algorithm has a higher complexity than the dead reckoning algorithm, requiring a "strong" application to handle it [6], which renders it unfeasible in large CVEs.

The Georganas group introduces the original concept of "interaction streams" [4], "each consisting of a burst of update messages with a final and a critical update message". The concept is supported on a proprietary multicast transport protocol, the Synchronous Collaboration Transport Protocol (SCTP), which is ACK based and adapted to the transmission of key update messages. In this model, the mechanisms to achieve state consistency do not use synchronous information like, for instance, clock or time-stamping information, which main constrain its applicability to large CVEs.

There are other approaches to shared state consistency, based on synchronisation mechanisms. One of the more significant mechanisms is the bucket synchronization mechanism [7], which is used in the MiMaze multi-player game [7]. In this case, time is divided into fixed length sampling periods and a bucket is associated with each sampling period. All application data units (ADU) received by an application are stored in the bucket corresponding to the current interval. When the application has to deliver an updated global state, it computes all ADUs available in the current bucket [7]. This algorithm needs global clock mechanisms like NTP [8] and the synchronization delay is computed on ADU's reception. If the network delay is greater than a given threshold the ADU is dropped. MiMaze uses an unreliable communication system, based on RTP [9] over UDP/IP multicast [10].

This paper presents a set of state transmission mechanisms to be used in the ARMS middleware platform [1]. The objectives of the proposed mechanisms, hereafter referred to as state transmission framework (STF), are to extend the applicability of the platform to collaborative virtual environments maintaining, at the same time, the platform's QoS features. Section 2 identifies the main requirements of collaborative virtual environments. Section 3 presents, to a considerable extent, the proposed state transmission framework. Section 4 describes the performance tests made to STF and analyses their results. The conclusions and guidelines for further work are presented in Section 5.

2. CVE's requirements

CVE applications have specific requirements in terms of scalability, interaction and consistency. The QoS characteristics that are relevant to these requirements are reliability, losses, delay and delay jitter. Additionally, application factors like data heterogeneity, frequency of events, synchronisation delay, number of participants and playout time (display frequency) [11] may play an important role in the behaviour of CVEs.
The following sub-sections briefly discuss some of these requirements and associated QoS characteristics. In turn, these have led to the development of several mechanisms that have been included in the ARMS middleware platform [1], which build on a set of extensions to the CORBA Event Service, providing native multicast communication, various reliability levels, congestion control and jitter suppression, with the aim to achieve QoS adaptability. These new mechanisms extend the ARMS platform with CVE-oriented QoS capabilities, and will be presented in Section 3.

2.1 Data heterogeneity

Typically there are many different types of data exchanged in CVE's [2]: real-time audio and video data, scene description data, typical 2D data, control data and state or update data. In addition to dealing with various types of data, continuous distributed interactive media can change their state in response to user operations as well to the passage of time [5]. A broad variety of applications use this kind of media, such as multi-user virtual reality (VR), distributed simulations, networked games and computer-supported co-operative work (CSCW) applications.

2.2 State synchronisation

All participants in a session must be synchronised in the same media state, i.e., the distributed shared state must be consistent. Different media have different state classifications and different state synchronisation needs. The state could be a simple change in a component. In the other extreme, it could be a bulk of data for latecomers or for re-synchronization. Some states may be essential and others may be redundant. State synchronisation must take the media type into consideration. Essentially, this is an issue for the applications' environment model, which must use the best solutions to deal with specific media.

2.3 Delay and jitter

Regular collaborative update messages have stringent delay requirements in order to maintain the shared state of components. Some studies [12] suggest that CVEs must have an end-to-end delay less than 100 msec. Others [13] consider 200 msec as an acceptable delay. In addition to delay, delay jitter also affects update messages. As shown in [13], a session with 10 msec delay and considerable jitter results in a perceived quality that can be as bad as one with 200msec delay and no associated jitter.

2.4 Reliability

CVEs for distributed interactive media require that all participants must receive state changes. Due to their specificity there are some states that are time critical and described by small amounts of information, while others are generally non-time-critical and require large amounts of information for their description. Therefore, there is the need for different levels of reliability when exchanging these types of updates: minimal reliability (possibly with loss detection) for the former, and full reliability for the latter.
In a CVE, the last state of a shared object is the most crucial data [4]. These messages must be sent with a high reliability level, whilst regular messages can be sent with a different, lower reliability level as, for instance, best effort.
Basically, there are two forms of achieving reliability: by using a reliable transport protocol or by using network-aware applications. In the context of interactive media applications, loss detection and reliability become more complex since there is no longer a single linear namespace for objects and since some objects are persistent [14]. So, a single transport protocol is unlikely to be sufficient, as observed in [14]. Additionally, many authors [15, 16, 17, 18] have concluded that application level framing is a requirement too, in this application context.
Both approaches to achieve reliability should be usable with a framework like a framing protocol. This framework could be a middleware platform with a proper interface to the application level, which captures the common aspects of a media class, and provides access to reliable transport protocols.

2.5 Other requirements

In [14], it is observed that many applications need structured application data unit (ADU) names, a simple mechanism for packet loss detection, a means of distinguishing different types of data, a means of identifying participants, and a time stamping mechanism.
All of these requirements add great complexity to the application level. Placing some of these capabilities in the middleware gives applications the ability to concentrate on specific functionalities, to enforce different adaptation policies and to interact with other components in the system in order to ensure fairness and other global properties.

3. ARMS state transmission mechanisms

The state transmission mechanisms presented in this paper were developed in the context of the Augmented Reliable Multicast CORBA Event Service (ARMS) middleware platform [1], that provides an end-to-end communication framework with QoS-matching capabilities. ARMS offers a set of QoS-related mechanisms for reliability guarantee in multicast environments, congestion control and jitter control. The QoS management process is supported on object-based monitoring and adaptation functions. The platform has specific objects for loss and jitter monitoring. The general architecture of the ARMS platform (Figure 1) includes an ARMS QoS API that provides access to the QoS features of the reliable multicasting services, and to the standard CORBA Event Service. Additionally, the architecture includes the STF API, which will be described in the remaining part of this section.

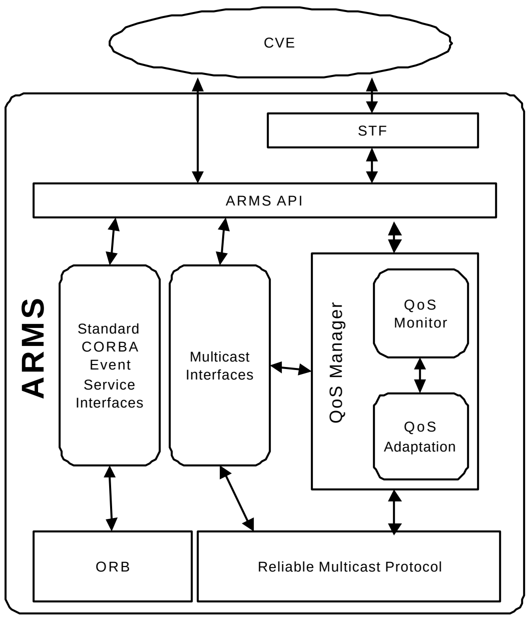


Figure 1 – ARMS architecture

3.1 State transmission framework

The STF API is a Java-based middleware especially designed for the transmission of state changes in virtual reality environments, which takes into account the requirements of state transmission in these types of applications. STF is an object-oriented framework integrated in the ARMS middleware platform (Figure 1) that supports state transmission and reception, a late join protocol – a process by which a client can join a ongoing interaction session and reconstruct the current global state, virtual world partitioning, and time synchronization. This paper discusses mainly the transmission and reception of states. The STF API comprehends a set of interfaces that expose methods that allow for the control various capabilities. Figure 2 illustrates the general architecture of STF, in UML.

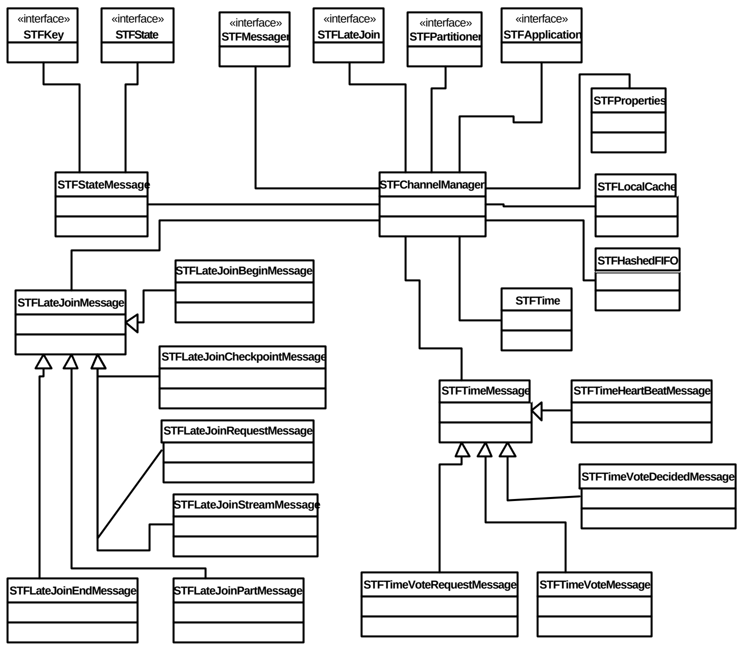


Figure 2 – Diagram of STF UML classes

3.2 State categories

In STF, each state may be classified into various categories, according to the way it must be transmitted to and from the network and the way it must be treated with respect to the late join protocol. In terms of redundancy, a state may be classified as follows.

- Redundant state – A redundant state may fail to reach the destination, because it may be overwritten in the buffers by more recent states of the same stream. A stream made of redundant states always has the more recent states transmitted, in detriment of other, older states.
- Essential state – An essential state always reaches the destination. A stream made of essential states always has all its messages transmitted.

In terms of volatility, a state may be classified as follows.

- Volatile state – A volatile state may fail to reach the destination, even if it is not overwritten by any state in the buffers. A stream made of volatile states may have none of its states transmitted.
- Non-volatile state – A state that has not the volatile characteristic.

With respect to the late join protocol, a state may be classified as follows.

- Independent state – An independent state object completely describes the state of the object it refers to; thus, in a late join process it is sufficient to recover the last transmitted independent state.
- Cumulative state – A cumulative state object does not completely describe the state of the object it refers to, but only a state change. With this kind of states, it is necessary to recover a complete set of transmitted state objects to execute a late join process.

Excluding the late join classification as independent or cumulative, state categories lead to four possible combinations. Of these only three are meaningful:

- Essential and non-volatile – All states reach the destination
- Redundant and non-volatile – The latest state reaches the destination
- Redundant and volatile – The states are not guaranteed to reach the destination

The fourth possible combination, essential and volatile, must not be used, as it is obviously impossible to guarantee that a volatile state always reaches its destination, as required by essential states.

3.3 State interaction streams

What exactly is an object state is highly dependent on the specific virtual reality application that is using STF. As such, STF makes as few assumptions as possible about the state object. In fact, STF considers as state object any object that can be serialised and implements the STFState interface. This interface contains methods that permit STF to find out the special characteristics of each state object.
Similarly to the work presented in [4], the STF API divides the transmitted states into interaction streams, identified by a unique key that corresponds to a particular virtual world entity. Each interaction stream is made of state messages that include the state objects corresponding to that entity ordered by timestamp. Figure 3 illustrates this concept.

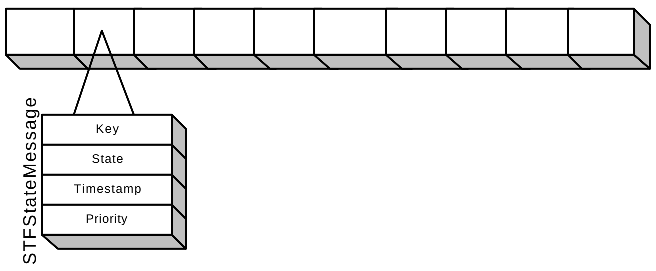


Figure 3 – State interaction streams

Using STF, applications can mix any types of states in the same stream, so we may have, for example, redundant states and some essential states in the middle, volatile and non-volatile states, etc. The streams are automatically generated by the API and are used to handle redundant and volatile states, so that STF may know which states to discard, if needed. A stream is made of *STFStateMessage* objects. Each of these objects is a message to be transmitted by the API, and includes the following information: the *key* object that identifies the virtual world entity to which the state refers that, by consequence, identifies the interaction stream to which this message belongs; the *state* object to be transmitted – an object of a class that must implement the *STFState* interface; message *timestamp* information – generated automatically by the STF API when the message is received from the application; and *priority* information of the message, which is a very important QoS property introduce in ARMS by STF, used when two or more messages from different streams are to be transmitted to the network. The *priority* information can take up one of five values:

- PRIORITY_LOWEST – The lowest priority possible
- PRIORITY_LOW – Low priority
- PRIORITY_MEDIUM – Medium priority
- PRIORITY_HIGH – High priority
- PRIORITY_HIGHEST – Highest priority possible

3.4 Transmission of state messages

When the application wants to transmit a state object, it must construct an *STFStateMessage* object, containing the key – an object of a class that implements the *STFKey* interface, the state – an object of a class that implements the *STFState* interface, and the message priority information. The application then sends the message to be processed to the *STFChannelManager* using the *sendMessage* method of the *STFMessager* interface.

3.5 Reception of state messages

The state messages – instances of *STFStateMessage* – are received by the application through the methods of the *STFApplication* interface. The method normally used for message reception is the *receive* method, with the *STFStateMessage* object as a parameter.

3.6 Reception lag and time warp avoidance

A time warp happens when a message that should have been received before some other message is received after it. This may cause an inconsistency in the virtual world and must be prevented. To prevent this from happening, STF orders all messages by timestamp and delivers them all in that order. However, when multiple conference nodes are transmitting messages about the same object, the underlying communication levels do not usually guarantee the total order of these messages. To circumvent this problem, STF makes it possible to enable a reception lag time. When enabled, reception lag causes a message to remain in the buffers for a specified

minimal period of time since it has been passed to STF by the emitter application. In this way the buffers are used to order incoming out of order messages such that they do not cause a time warp.
The reception lag may degrade application performance, and even endanger virtual world consistency if not used wisely. So, this feature must be used with extra care. Reception lag is depicted in Figure 4.

3.7 Time warp detection

With or without reception lag, time warps are possible when transmitting data about the same entity in two or more conference nodes. When a time warp happens, STF detects this and delivers the time-warped message to the application using a special method of the *STFApplication* interface – *timeWarpReceive*. The application then has the chance to consider the message, ignore it or do whichever action it deems adequate to the situation.

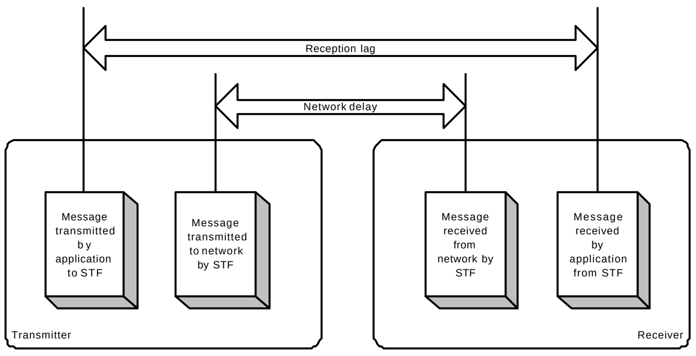

Reception lag
Network delay
Message transmitted by application to STF
Message transmitted to network by STF
Message received from network by STF
Message received by application from STF
Transmitter
Receiver

Figure 4 – Reception lag components

4. Tests made to STF

In order to analyse the performance of the state transmission framework, a series of tests designed to evaluate the behaviour of this API were made, covering common situations in virtual reality applications. Due to the extension of the API, which comprises time synchronization, late join protocols, various transmission and reception options, and partitioning of virtual world entities it was, in practice, impossible to test all possible configurations. Thus, only tests related to common state transmission configurations were made, with the aim of discovering the limits of the API and assessing its behaviour under various circumstances.

4.1 The testbed

The tests were made using three Pentium III 733Mhz-500Mhz computers (A-TEJO, B-CONCHA and C-SADO), with 128 MB RAM and Windows 2000 Professional, directly connected to a network switch by 100Mbit Ethernet full duplex links. All three computers had the Java Software Development Kit version 1.3, from Sun Microsystems [19], installed. The state transmission framework ran as part of the CONCHA conference controller system version 2.0 [20], which used Java 1.3, JSDT 2.0 [21] and ARMS 1.0 [20]. All three computers were synchronized using NTP through the installation of the NTPTime client for WindowsNT, adapted to work under Windows2000 [22]. All three computers used the same NTP server for time synchronization.

4.2 Tests description

The objective of the tests was to measure the total message delay and also the message transmission throughput under different conditions, so as to conclude about STF's efficiency.
Two sets of tests were performed. The first set addressed the effect of increasing message transmission rate (increased throughput) on the total transmission delay and the redundant message discarding. These tests were used to conclude about the practical limits of STF under stress conditions.
The second set of tests addressed the effect of increasing message sizes (through the increase in state sizes) on the total transmission delay and the redundant message discarding, using a fixed message transmission rate. Both sets of tests used three different streams of states, with the purpose of simulating a simple but representative situation of mixed streams with different state characteristics:

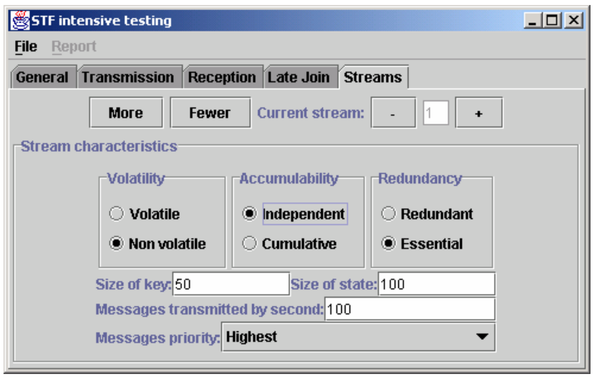
- Stream 1: redundant, volatile and independent states;
- Stream 2: redundant, non-volatile and independent states;
- Stream 3: essential, non-volatile and independent states.

All state messages from all streams had the same priority – Highest.

The first set of tests – the message rate tests – used a state size of 22 bytes and a key size of 6 bytes, totalling 28 bytes. This is enough to transmit three-dimensional position and rotation information, which is sufficient for many applications, though not all. The test started by transmitting five messages per second (msg/s) in each stream. This was increased by 5 msg/s in each consecutive run. The test was set to stop at 400 msg/s for each stream, which amounted to a total of 1200 msg/s. Additionally, the test was programmed to stop as soon as the total message transmission delay – comprising the message transmission by the sending application to STF and the message delivery to the receiving application by STF – would reach one second, a value that was considered unacceptable. In the second set of tests, state messages of 22 bytes were first used, the state size being incremented by 100 bytes in each consecutive run. In both sets of tests, each individual test ran for 20 seconds.

4.3 The test application

A test application running as part of the CONCHA system version 2.0 [20,23] was created with the specific purpose of performing the STF API tests. The application enables the user to specify all properties of the STF session, such as underlying communication properties, transmission and reception lag control, late join protocol control, time synchronization settings and time warp detection. Figure 5 presents a screenshot of this application, where the setting of stream state transmission properties is visible.



Current stream: 1

Stream characteristics

Volatility: Volatile [✓ Non volatile]

Accumulability: [✓ Independent] Cumulative

Redundancy: Redundant [✓ Essential]

Size of key: 50

Size of state: 100

Messages transmitted by second: 100

Messages priority: Highest

Figure 5 – Screenshot of the test application

This application allows for a completely automated testing process, through an option in the application's menu ("Start automatic testing") that executes the tests earlier discussed in this paper. It also allows for more specific testing of a large variety of situations under which STF may be used. With this application, it is possible to test most of STF's capacities without having to build a complete multiuser CVE. However, the authors plan to build such an environment in the near future in order to test and evaluate other STF's features that this application does not test properly, such as some of the features of the late join protocol.

4.4 Tests' results and analysis

Figure 6 identifies the four message probing points used to gather delay and throughput data. Point (A) corresponds to the sending application interface with STF. Messages passing through this point are shown in Figure 9 as *processed* messages. Point (B) is the sending STF interface with the communications layer. Messages passing through this point are shown in Figure 9 as *transmitted* messages.

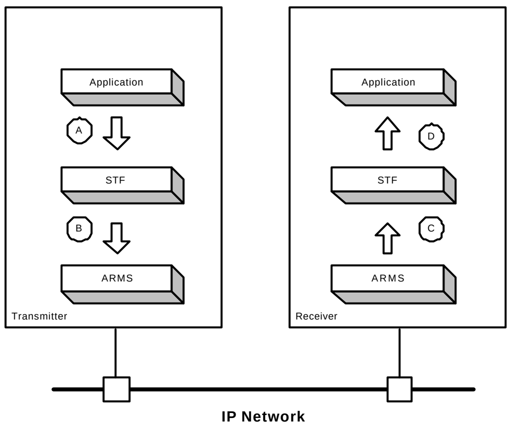


Figure 6 – Probing points

The obtained results have shown that these are the same as the messages that pass through the third message probing point (C), which correspond to the messages received by STF from the underlying communications layer at the receiving node. Thus, these messages are not represented in Figure 9, since the data are the same as the transmitted messages data. The fourth and final message probing point corresponds to the receiving application interface with STF (D). The corresponding messages are shown in the graphs as *received* messages.

The STF latency is the time that messages take since they travel from the probing point (A) to the probing point (D). Latency was measured as a function of message throughput (Figure 7) and as a function of message size (Figure 8).

Figure 7 shows that the average total message delay, that is, the average latency is less than 20 ms for the vast majority of tests and is always less than 25 ms. It is noteworthy to say that this latency includes all STF and ARMS overheads (marshalling/demarshalling, encoding/decoding of the reliable multicast protocol packet, etc). Additionally, there was no problem in reaching the target value of 400 msg/s per flow.

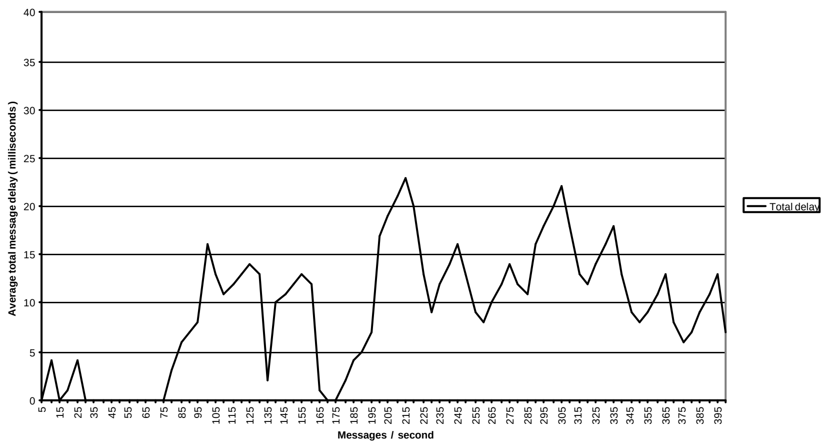


Figure 7 – Latency as a function of state rate

Figure 8 shows that, similarly to the latency versus state rate case, the average latency is generally less than 20 ms even when the states' size is considerably high, well above the 1400-byte limit that implies ARMS fragmentation/reassembling overhead.

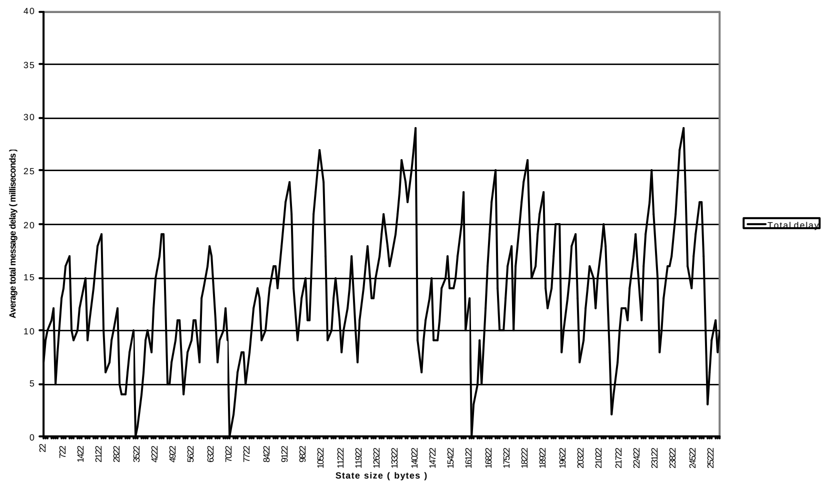


Figure 8 – Latency as a function of state size

Figure 9 shows that when the network traffic increases there is, in fact, an efficient utilization of network resources by STF, discarding redundant and volatile messages as needed, but keeping the essential ones. This maintains network traffic at acceptable levels even when STF reaches full state transmission capacity, without losing the consistency of the shared global virtual world state.

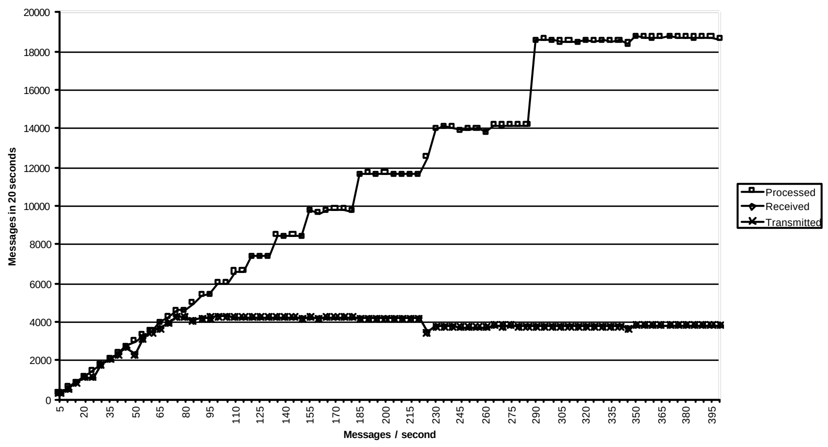


Figure 9 – Processed, transmitted and received messages as a function of state rate

5. Conclusions and guidelines for further work

Middleware platforms must have specific characteristics in order to adequately support collaborative virtual environments. In addition to good reliability, delay and jitter characteristics, it is essential that state synchronisation is efficiently guaranteed and maintained.

In this paper, a proposal of a set of mechanisms that provide such synchronisation was made. These mechanisms, collectively referred to as a state transmission framework, were implemented and subject to functionality and performance testing in the context of a QoS-aware middleware platform named ARMS, developed by the authors in a previous project.

After a presentation of the main features of the framework, some of the tests' results were presented and discussed. In addition to validating concepts so important as the state redundancy and state volatility concepts, the tests have clearly shown that the implemented prototype has good performance in terms of throughput, latency and efficiency in the use of both processing and network resources.

Subsequent phases of this work will address further testing, namely scalability testing. Additionally, future work will try to optimise the integration with the ARMS platform, with emphasis on the exploration of some of its QoS capabilities such as multiple reliability levels.

Acknowledgement

This work was partially financed by the Portuguese Foundation for Science and Technology, FCT.

References

[1] João Orvalho, Fernando Boavida, "Augmented Reliable Multicast CORBA Event Service (ARMS): a QoS-Adaptive Middleware", in *Lecture Notes in Computer Science, Vol. 1905: Hans Scholten, Marten J. van Sinderen (editors), Interactive Distributed Multimedia Systems and Telecommunication Services,* Springer-Verlag, Berlin Heidelberg, 2000, pp. 144-157. (Proceedings of IDMS 2000 – 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services, CTIT / University of Twente, Enschede, The Netherlands, October 17-20, 2000).

[2] S. Singhal, M. Zyda. "Networked Virtual Environments Design and Implementation", ACM press, New York, 1999.

[3] Martin Mauve. "How to Keep a Dead Man from Shooting", in *Lecture Notes in Computer Science, Vol. 1905: Hans Scholten, Marten J. van Sinderen (editors), Interactive Distributed Multimedia Systems and Telecommunication Services,* Springer-Verlag, Berlin Heidelberg, 2000, pp. 144-157. (Proceedings of IDMS 2000 – 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services, CTIT / University of Twente, Enschede, The Netherlands, October 17-20, 2000).

[4] Shervin Shirmohammadi and Nicolas D. Georganas. "An End-to-End Communication Architecture for Collaborative Virtual Environments", Computer Networks Journal, Vol.35, No.2-3, Febr. 2001, pp.351-367.

[5] Martin Mauve, "Consistency in Continuous Distributed Interactive media", Technical Report TR-9-99, Reihe Informatik, Department for Mathematics and Computer Science, University of Mannheim, November 1999.

[6] Martin Mauve, "Distributed Interactive Media", Ph.D. Thesis, University of Mannheim, Germany, September 2000.

[7] L. Gautier and C. Diot, "Design and evaluation of MiMaze, a Multiplayer Game on the Internet", IEEE Multimedia System Conference, Austin, June 28- July 1, 1998.

[8] David L. Millis, "Network Time Protocol (version 3) specification, implementation", Request For Comments 1305, IETF, March 1992.

[9] Schulzrinne, Casner, Frederic, Jacobson, "RTP: A transport Protocol for Real-Time Applications", revision of RFC 1889, Internet-Draft (draft-ietf-avt-rtp-new-04.ps), June 25 1999.

[10] S. E. Deering, "Multicast Routing in Datagram Internetwork", Ph.D. dissertation, Standford University, December 1991.

[11] Dimitrios Makrakis, Abdelhakim Hafid, Farid Nait-Abdesselem, Anastasios Kasiolas, Lijia Qin, "Quality of Service Management in Distributed Interactive Virtual Environment", Progress Report of DIVE project.
http://www.mcrlab.uottawa.ca/research/QoS_DIVE_Report.html

[12] M. M. Wloka, "Lag in Multiprocessor VR", Presence: Teleoperators and Virtual Environments (MIT Press), Vol 4, Nº 1, Spring 1995.

[13] K. S. Park And Robert V. Kenyon, "Effects of Network Characteristics on Human Performance Collaboration Virtual Environment", IEEE International Conference on

Virtual Reality (VR '99), Houston, Texas, March 1999.

[14] Colin Perkins and Jon Crowcroft. "Notes on the use of RTP for shared workspace applications", ACM Computer Communication Review, Volume 30, Number 2, April 2000.

[15] J. Crowcroft, L. Vicisano, Z. Wang, A. Ghosh, M. Fuchs, C. Diot, and T. Turletti, "RMFP: A reliable multicast framing protocol", March 1998. Work in progress (Internet draft).

[16] B. DeCleene, S. Bhattacharaya, T. Friedman, M. Keaton, J. Kurose, D. Rubenstein, and D. Towsley, "Reliable multicast framework (RMF): A white paper", March 1997.

[17] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang., "A reliable multicast framework for light-weight sessions and applications level framing", *IEEE/ACM Transactions on Networking*, December 1997.

[18] M. Handley and J. Crowcroft, "Network text editor (NTE): A scalable shared text editor for the Mbone", In *Proceedings ACM SIGCOMM'97*, Cannes, France, September 1997.

[19] JavaSoft, www.javasoft.com

[20] João Orvalho, Luís Figueiredo, Tiago Andrade, Fernando Boavida, "A platform for the study of reliable multicasting extensions to CORBA Event Service", in *Lecture Notes in Computer Science, Vol. 1718: Michel Diaz, Philippe Owezarski and Patrick Sénac (editors), Interactive Distributed Multimedia Systems and Telecommunication Services*, Springer-Verlag, Berlin Heidelberg, 1999, pp. 107-120. (Proceedings of the 6th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services, IDMS'99, IEEE, LAAS-CNRS, ENSICA, Toulouse, France, October 12-15, 1999).

[21] Java Shared Data Toolkit (JSDT), SUN Microsystems, JavaSoft Division, http://java.sun.com/people/richb/jsdt.

[22] The NTPTime client for the network time protocol, http://home.att.net/~Tom.Horsley/ntptime.html.

[23] João Orvalho, Tiago Andrade, Luís Figueiredo, Fernando Boavida, "CONCHA – CONference system based on java and corba event service CHAnnels", Proceedings of SPIES's symposium on Voice, Video, and Data Communications conference on Quality of Service Issues Related to Internet II, Boston, MA, USA, September 19-22, 1999.