Support for High Performance Real-time Collaboration within the NaradaBrokering Substrate

Shrideep Pallickara[1], Hasan Bulut[1], Pete Burnap[2], Geoffrey Fox[1] , Ahmet Uyar[3], David Walker[2]
spallick@indiana.edu, hbulut@indiana.edu, P.Burnap@cs.cardiff.ac.uk, gcf@indiana.edu , auyar@syr.edu and David.W.Walker@cs.cardiff.ac.uk

Community Grids Lab, Indiana University[1]
School of Computer Science, Cardiff, UK[2]
Department of Electrical Engineering & Computer Science, Syracuse University[3]

Abstract

The requirements for collaborative services, especially pertaining to order and delivery, are quite different compared to traditional distributed applications. The NaradaBrokering messaging substrate enables scalable, fault-tolerant, distributed interactions between entities, and is based on the publish/subscribe paradigm. The substrate also incorporates support for Grid and Web Service. More recently, we have incorporated services within the substrate which enable us to facilitate richer collaborative interactions. In this paper, we outline our rationale for incorporating these services and how these services interact with each other. We have conducted experiments related to profiling these services, and also on the performance and scaling of the messaging substrate. Our experimental results demonstrate that the substrate can indeed be used in scenarios where performance and scalability requirements are stringent.

Keywords: distributed messaging, collaborative services, a/v conferencing, replay services, buffering, scalable systems

1. Introduction

A collaborative system can be characterized as a system where a group of users have come together with the intent to exchange (and share) data, state transitions and actions initiated by participants. The data shared could be text, graphics, shared displays or multimedia content. Applications typically manage the type of data that is received by using appropriate content handlers. At its very core the fundamental problem within collaborative systems is one of disseminating the right content to the right participants. Furthermore, since the participants in a collaborative session are distributed over a wide area network, the underlying infrastructure supporting collaboration needs to cope with the complexities of communications, network failures and fluid group memberships.

Approaches to collaboration have tended to use IP Multicast to deal with the content distribution problem. Multicast provides a powerful, elegant and flexible framework for implementing collaborative systems. Here, participants agree upon a multicast group and collaborate by exchanging data over this group; the system relies on MBONE to manage this data interchange.

In this paper we suggest that a far more powerful framework for collaboration is the publish/subscribe paradigm. In publish/subscribe systems the routing of messages from the publisher to the subscriber is within the purview of the message oriented middleware (MoM), which is responsible for routing the right content from the producer to the right consumers. Subscribers register their interest in content through subscriptions, publishers are responsible only for the generation of content. Publish/Subscribe systems thus provide a clear decoupling of the message producer and consumer roles that interacting entities might have. This is especially useful if there are a large number of potential consumers for a given message. In such cases a producer need not keep track of the large number of consumers that a message could potentially be routed to: the middleware performs this function for the publisher.

Support for high performance collaboration needs to address several issues which we enumerate here.

1. Scaling: The underlying infrastructure should cope with the presence of a large number of entities while facilitating the management of a large number of collaborative groups.
2. Support for complex collaborative schemes: Here, entities may be interested in receiving notifications

under very specific scenarios. This would involve notification of membership changes or the availability of content that meets a specific constraint set by the interested entity.

3. Support for timing based services: This includes support for high resolution timestamps, buffering and time-based ordering of messages. In the case of time based ordering the timestamps in individual messages should also cope with clock skews and synchronization problems.
4. Recovery and Replay Services: It should be possible for late (or recovering) participants to retrieve missed messages. Furthermore, replays should be able to preserve time spacing between messages; if so desired.
5. Performance: The infrastructure should be able to provide consistent throughput and high-performance while supporting each of the aforementioned scenarios.

We investigate these issues in the context of our system: NaradaBrokering. NaradaBrokering is a content-based publish/subscribe system and has been in the open source domain for the past 3 years. We have incorporated several services within NaradaBrokering to allow it to cope with the stringent requirements intrinsic in high performance collaborative systems. In this paper we demonstrate that the NaradaBrokering substrate provides a powerful, generalized framework for enabling high-performance collaborative applications.

The remainder of this paper is organized as follows. In section 2 we provide an overview of the NaradaBrokering substrate, here we also contrast the advantages of the publish/subscribe paradigm over hardware Multicast. In section 3 we discuss the various services available within the NaradaBrokering substrate which can be leveraged by high performance collaborative applications. We have conducted experiments profiling the performance of the substrate and its constituent services for supporting collaborative applications with some of these experiments being performed in trans-Atlantic settings. We include these results within the relevant subsections. In section 4 we present an overview of the related work. Finally, in section 6 we outline our conclusions and future work.

2. NaradaBrokering Substrate

NaradaBrokering [1-7] is a distributed messaging infrastructure based on the publish/subscribe paradigm, and provides two closely related capabilities. First, it provides a message oriented middleware (MoM) which facilitates communications between entities (which includes clients, resources, services and proxies thereto) through the exchange of messages. Second, it provides a notification framework by efficiently routing messages from the originators to only the registered consumers of the message in question. This dissemination constraint holds true irrespective of the size of the broker network or the number of clients within the system. The smallest unit of this *substrate* which intelligently processes and routes messages, while working with multiple underlying communication protocols is referred to as as a *broker*. The NaradaBrokering software has been in the open source domain for the past 3 years. The current code-base comprises 1100 classes with approximately a quarter million lines of code.

Communication within NaradaBrokering is asynchronous and the system can be used to support different interactions by encapsulating them in specialized messages (also referred to as *events*). These specialized messages can encapsulate information pertaining to transactions, data interchange and system conditions. NaradaBrokering places no constraints either on the size, rate or scope of the interactions encapsulated within these messages, or on the number of entities present in the system.

NaradaBrokering relies on software multicast for communications; this obviates the need for MBONE which is required for multicast communications. It should however be noted that the substrate can leverage hardware multicast if it is available. The NaradaBrokering substrate provides support for transport protocols such as TCP, Parallel TCP, UDP, Multicast, HTTP and SSL; it also facilitates communications across NAT and firewall/proxy boundaries.

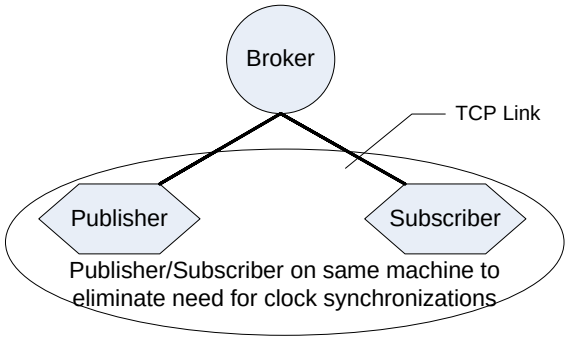


Figure 1: Experimental setup for measuring communication latencies

To enable the reader to get an idea of the costs involved in communications within the substrate, we now report

results pertaining to communications within the NaradaBrokering substrate. The experimental setup for our measurements is depicted in Figure 1. The results are depicted in Figure 2, where each point in the delay-curve corresponding to the average of 50 messages. The standard deviation-curve reports the deviation in these delays. Note that the numbers reported here correspond to two-hops, from the producer to the broker, and from the broker to the producer. The per-hop latency in cases up until 4KB is around 1 millisecond (transit delay corresponds to traversal from publisher-broker-subscriber). The results reported here are for communications using TCP.

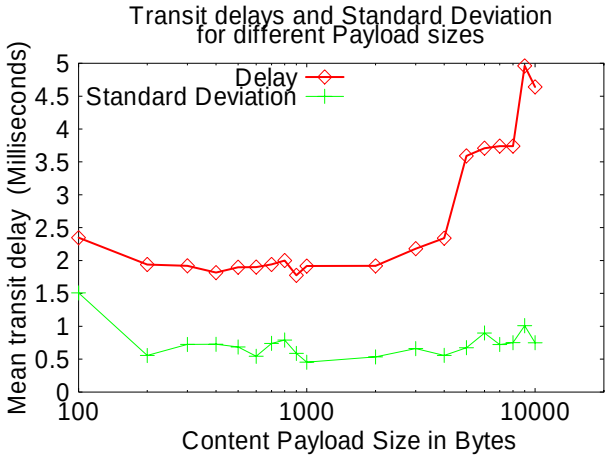


Figure 2: Average delays and standard deviation for message samples

NaradaBrokering allows clients to register their subscriptions (interest in the content of messages) using a variety of formats. The subscriptions can be in the form of String, Integer, Long and <tag, value> based topics, or in the form of XPath, SQL and Regular expression queries. Support for this variety of subscription formats also implies richer collaborative interactions since actions may be triggered only under very precise conditions. The complexity of managing these subscriptions and routing relevant messages is delegated to the middleware substrate. Since the individual entities do not need to cope with the complexity of constraints, this in turn facilitates easier development of collaborative applications which enable these complex interactions.

Typically, the number of managed subscriptions increases with the number of collaborative groups and entities present within the system. Depending on the complexity of interactions the subscription formats may vary. To give an idea of the cost involved in managing various subscription formats, we include some results from our measurements with Integers, Strings and <tag-value> pair based topics. To contrast the cost involved in managing richer subscription constraints we also include results for Regular expressions. For every case the number of subscriptions is varied from 20,000 to 100,000; and the results depict the time to compute destinations for a given message from this set of subscriptions. The results were measured on a machine (1GHz, 256MB RAM) running the process in a Java-1.4 Sun VM with a high-resolution timer for computing delays.

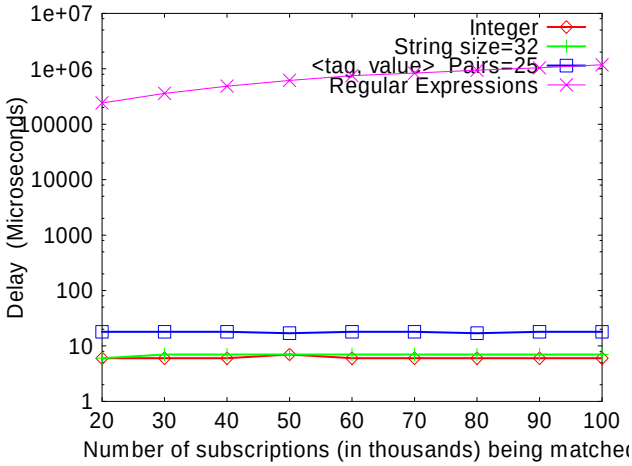


Figure 3: Costs involved in different subscription constraints

The richer the constraints, the greater the CPU-cost associated with the matching process, which computes destinations for a message. As can be seen, in Figure 3, the average cost for matching increases progressively (Integer to String to Tag-Value to Regular Expressions in that order) as the complexity of the matching increases. For Integer matching the average cost was around 6 microseconds, while for String based matching it was 7 microseconds. The costs associated with <tag,value> based matching when there are approximately 25 <tag,value> pairs was around 18 microseconds. Finally, in the case of Regular expressions the cost varied from 242 milliseconds for 20,000 subscriptions to 1.178 seconds for 100000 subscriptions. The results also demonstrate the feasibility of using software multicast for communications.

2.1 Support for scaling within the NaradaBrokering substrate

In this section we investigate the limits of the brokering substrate in supporting a large number of multi-media clients. For the purposes of our experiment we have recorded an audio and a video stream for 2 minutes. The audio stream is based on the 64 kbps ULAW codec; the size of each package is 252 bytes and is issued once every 30 milliseconds. For the duration of the experiment there were 4100 packages without any silence period within the stream. Note that this is telephone-quality audio and is

quite widely used in videoconferencing sessions over the Internet.

We also recorded the video stream of a speaking participant in a video conferencing session setting. This is an H.263 stream with 15 frames per second. The video encoder encoded a frame every 66ms. Although the average bandwidth was 280 kbps; the bandwidth fluctuated mostly between 250 kbps and 310 kbps. For the duration of the experiment there were 1800 video frames which were transmitted during 2 minutes, and a total of 5610 packages. Note that the video encoder was dividing the frames that have more than 1 KB of data into multiple packages. The average length of individual video packages was 740 bytes. The transmitter sent one full picture update frame every 60 frames or every 4 seconds.

We first conducted tests to evaluate the performance and the limits of a single NaradaBrokering broker. These experiments were performed on an 8 node Linux cluster with a gigabit network switch. All nodes had dual-CPUs (Intel Xeon 2.4GHz) and 2GB of memory. The runtime environment for all components involved in the experiment is JDK 1.4. Please note that in these experiments each meeting is designed as a single speaker meeting with one speaker and many listeners. Furthermore, when calculating the latencies and accompanying jitters, we ignored the first 100 packages to compensate for start up costs. The Jitter J is computed based on the formula outlined in the RTP [8] specification – J = J + (|D(i-1, i)| - J)/16, where D(i-1, i) corresponds to the difference between the delay for ith RTP packet and the delay for the (i-1)th RTP packet.

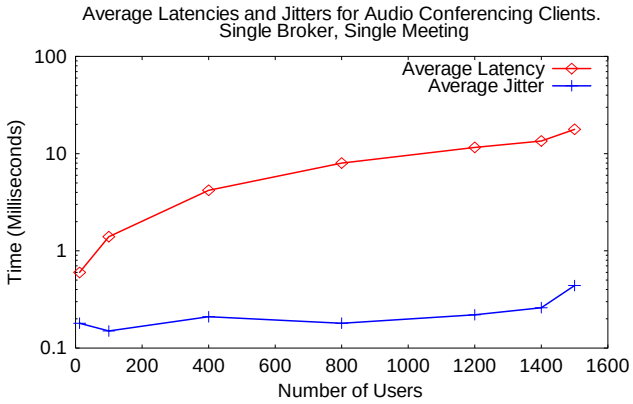


Figure 4: Average latency and jitter for audio conferencing clients

Figure 4 depicts the average latency and jitter for audio conferencing clients. The average latency value is less than 20 milliseconds (with jitter being very acceptable too) until the broker is overloaded. This overload takes place at 1600 users when the average latency jumps to 2.2 seconds. This was not depicted in the graph to ensure that the pattern up until 1500 clients was clearly visualized.

Figure 5 depicts the latency and jitter for video conferencing clients. Though the average latency for 900 clients is acceptable, the broker is actually overloaded when there are 500 participants. This is because the number of late arrivals which correspond to packets whose delay exceeds 100 ms is around 3% for 500 clients; a number at which we deem the broker to be overloaded.

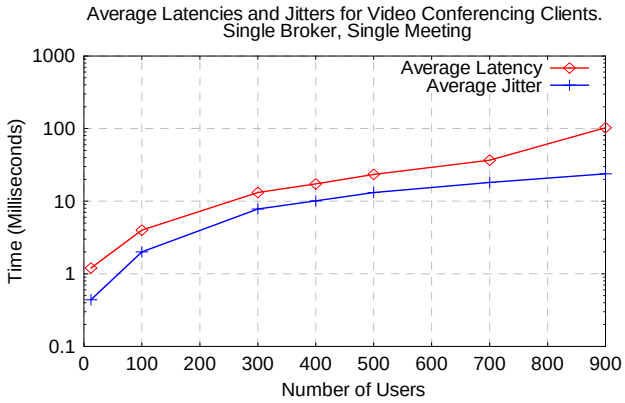


Figure 5: Average latency and jitter for video conferencing clients - Single Broker Single Meeting

Next, we evaluate the performance of the broker network with four brokers for a single video meeting.

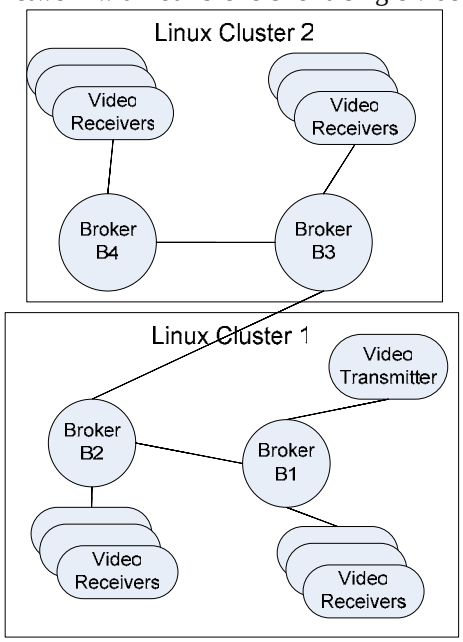


Figure 6: Experimental Setup for Multi-broker measurements

For the distributed broker tests, we used two Linux clusters, each having 8 nodes with the four brokers connected as depicted in Figure 6. The configuration of

nodes in the first cluster is 2.4GHz Dual Intel Xeon CPU and 2 GB of memory, while that of the nodes in the second cluster is 2.8 GHz Dual Intel Xeon CPU and 2GB of memory. The clusters had a gigabit network bandwidth among its nodes.

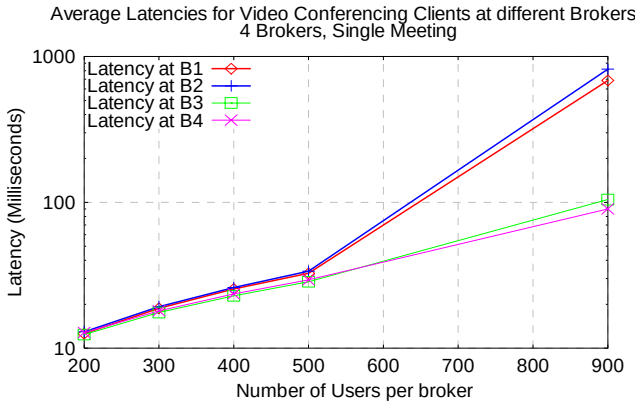


Figure 7: Average latencies for Video conferencing clients - 4 Brokers, Single Meeting

The receivers were evenly distributed among the brokers and across meetings. The latency values of brokers B1 and B2 are quite similar. Similarly, the latency values of B3 and B4 are very similar. Brokers B3 and B4 perform better than the first two brokers, because the machines in the second Linux cluster have superior CPU power. As the latency values show (Figure 7), adding new brokers increases the capacity of the broker network. In this case, since all brokers have quite similar computing power, each broker increases the capacity of the broker network almost linearly. For the first two brokers, the percentage of late arriving packages is 1.9% when there are 400 participants. Therefore, they can support up to 400 users. For the last two brokers, the packages arriving late are less than 1.0% for the same number of participants, thus supporting 400 users comfortably. In total, four brokers support close to 1600 participants in a single video meeting. Figure 8 and Figure 9 depict the average latency and jitter values respectively in multiple meeting settings involving the same broker network.

In summary, these experiments demonstrate that the NaradaBrokering broker network scales well in distributed settings when delivering streams to a high number of participants. The scalability of the broker network increases almost linearly with the number of brokers.

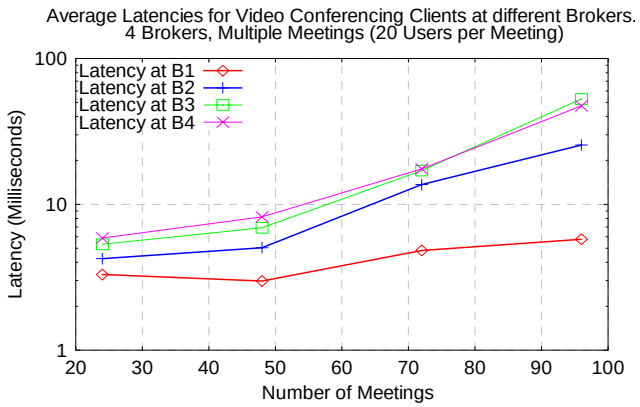


Figure 8: Average Latencies for Video Conferencing clients - 4 Brokers, Multiple meetings

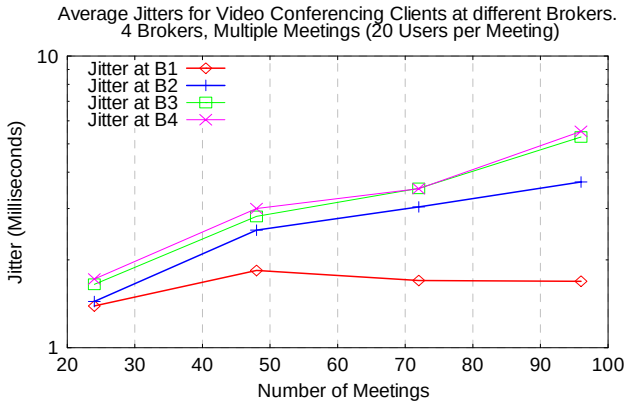


Figure 9: Average Jitters for Video conferencing clients -- 4 Brokers, Multiple meetings

Next, we investigate the delivery of audio/video streams to geographically distant clients. We had access to machines at three more locations, in addition to the two Linux clusters (in Bloomington, IN) that we used for the previous distributed broker tests. The other three sites were Syracuse University at Syracuse, NY, Florida State University (FSU) at Tallahassee, FL and Cardiff University in Cardiff, United Kingdom. These three sites had 90-100Mbps download bandwidths.

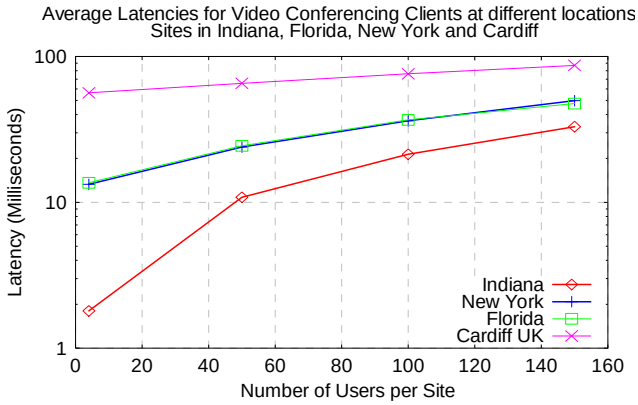


Figure 10: Average Latencies for Video Conferencing Clients - WAN

The broker was running at Indiana and an equal numbers of participants were running at the four other sites. A client running in the same site as the broker published the video stream on the broker. We measured the latencies and jitters at the clients. Please note that there can be a few millisecond discrepancies in latency values because of the difficulties in determining the exact clock differences between the transmitter and receiver. This test demonstrates that 150 video streams can be transferred between these four sites with acceptable transmission delays (Figure 10) and jitter (Figure 11).

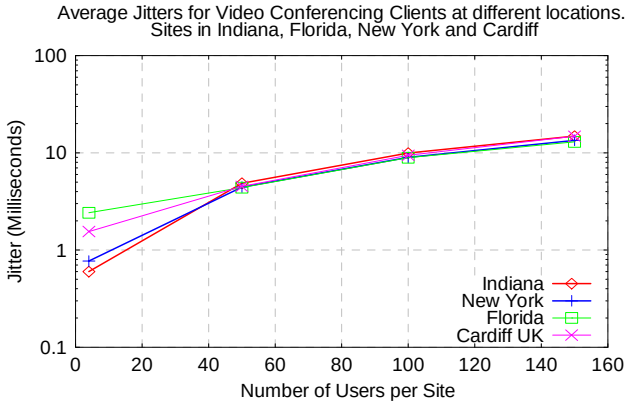


Figure 11: Average Jitters for Video Conferencing Clients - WAN

3. Services enriching collaboration within the NaradaBrokering substrate

In this section we outline services that we have incorporated into the NaradaBrokering substrate to facilitate richer collaborative interactions. The section elaborates on issues related to time ordering, spacing and the reduction of jitters during collaboration.

3.1 Ensuring consistent timestamps

Entities within a distributed system generate messages with timestamps based on their local clocks. Since the clocks on individual machines are out-of-sync with each other, these timestamps are not useful in time ordering these messages. Furthermore, on a given machine, clocks may run slower or faster than they should. What is needed is a scheme which accounts for these skews and provides consistent timestamps. To this end, we have implemented and incorporated the Network Time Protocol (NTP) [9] within the substrate.

NTP is one of the most widely used algorithms for ensuring consistent timestamps in distributed settings, and can achieve an accuracy of 1-30 milliseconds; this implies that timestamps generated at any point within a distributed system will be within 30 milliseconds of each other. However, this accuracy also depends on the roundtrip delay between the machine and the time service server. This difference in the communication delays between the host machine and the time server also contributes to the accuracy of the offset (which identifies how much the clock needs to be adjusted) that is computed. NTP achieves this accuracy by using filtering, selection and clustering, and combining algorithms to adjust the local time. In the NTP algorithm the rectifying-machine receives time from several time servers. The filtering algorithm selects the best samples obtained from a given time server. The selection and clustering algorithms then pick the best *truechimers* and discard the *falsetickers*. Finally, the combining algorithm computes a weighted average of the time offset of the best truechimers.

Consistent timestamps allow us to time order events since the timestamps generated at any entity within the system can now be assumed to be accurate within a certain range (1-30 msecs for NTP). Such precise timestamps can in turn enable time-ordering of events that have been issued by entities at disparate geographic locations. A problem that existed within a distributed publish/subscribe systems was the precise determination of when a subscription is to be considered active. This can be assuaged a great deal with the use of precise UTC timestamps available through the use of the NTP protocol. In the aforementioned scenario subscriptions will maintain a timestamp indicating when they were created, any events published after this timestamp should be considered for matching with (and delivery to) this subscription. Finally, we are currently investigating the use of these timestamps to ensure consistency within replicated shared contexts of distributed computations.

Figure 12 depicts the results from our measurements that were done with a machine in Cardiff, UK. The time servers that we used for this particular experiment involved stratum-1 and stratum-2 time-servers from Europe (specifically UK, France, Italy and Germany). The initial offset value computed by our NTP algorithm is 19235 milliseconds, which means that the system clock trails the real time by that amount. Changes in offset indicate the corrections that are taking place due to clock drifts. Note that we ensure consistency of timestamps returned by the service previously: negative offsets are not applied if they result in time traversing into the past.

Machine based in Cardiff UK
OS: Red Hat Linux 7.1 2.96-79
CPU: Pentium III, 1 GHz
Memory:1.5 GB
JVM Version: 1.4.1_01

initialization offset	
value	19235 msec
min	0 msec
max	5 msec
total change	527 msec

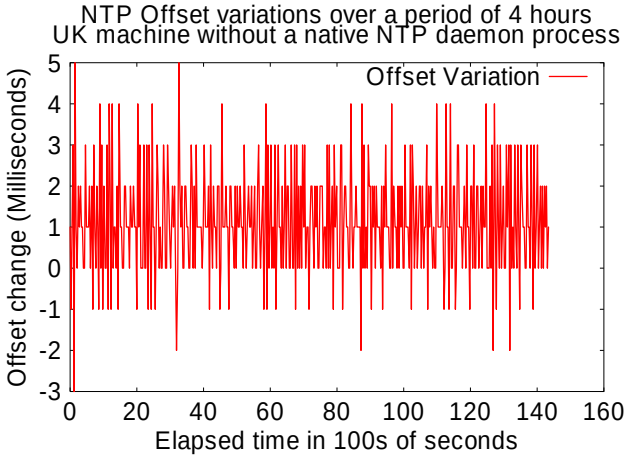


Figure 12: Change of Offset with Time (UK machine no NTP daemon)

Figure 13 depicts results from measurements performed on a machine in Indiana. In this test we used stratum-1 time servers that were based in US. The offsets computed vary between -1 and 3 milliseconds.

OS: Red Hat 3.4.2-6.fc3
CPU: Intel(R) Pentium(R) 4 CPU 1500MHz
Memory: 1 GB,
JVM Version: 1.4.

initialization offset value	544461 msec
min	0 msec
max	3 msec
total change	381 msec

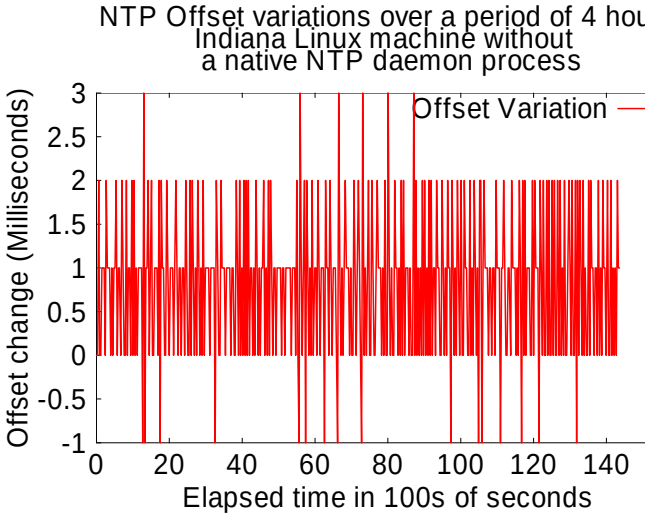


Figure 13: Change of Offset with Time (Indiana machine no NTP daemon)

OS: Red Hat Linux 7.3 2.96-110
CPU: AMD Athlon(tm) MP 1800+ 1. GHz
Memory:1034756 kB
JVM Version: 1.4.1_03

initialization offset value	1 msec
min	0 msec
max	1 msec

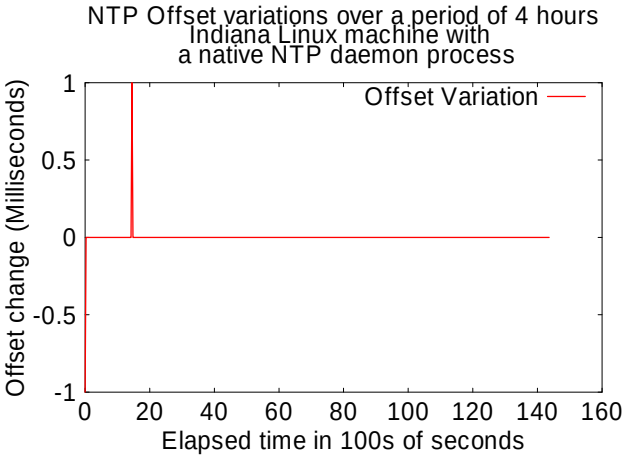


Figure 14: Change of Offset with Time (Indiana machine with NTP daemon)

Figure 14 depicts results from measurements performed on a machine with a native NTP daemon running on the machine; this ntpd daemon synchronizes its time with "time.nist.gov" time server. Note that the previous two

machines did not have this process running. Only 2 samples in this experiment are reported to be non-zero (-1 msec and 1msec) and all other values are reported to be consistent with the ntpd daemon. The initial offset computed is also 1 msec. This shows that ntpd daemon running on the machine and our time service are very consistent with each other.

3.2 High resolution timing services

To ensure that messages are time-stamped as accurately as possible we have also incorporated a high resolution timer service into the substrate. This high resolution timer works on the Windows, Linux and Solaris operating systems; here we have leveraged native libraries available on these systems along with the Java Native Interface to enable high resolution timers. On Solaris and Linux the *gettimeofday()* function is used to retrieve the current time in microseconds. It returns time since 1/1/1970; the resolution is hardware dependent and is usually around 1 microsecond. The `QueryPerformanceCounter` on Windows is used to get number of ticks; the number of ticks in one second is 3759545, which is around one per 279 nanoseconds. This returns ticks from the start of the machine, but does not have limitations as in the *getTickCount()* function, which rolls-over every 49 days. Note that this gives us better results than relying only on the Java call. The resolution of the System.currentTimeMillis() on Windows is around 15 milliseconds and 1 millisecond on Linux. We have measured the resolution of the high resolution timer to be around 3~4 microsecond.

3.3 Time buffering service

Jitter is considered to be one of the most important Quality of Service (QoS) measurements within A/V collaborative systems. In the case of audio streams, high jitter values can cause voice breaks while in the case of video streams high jitters may cause degenerations in the image quality. In order to overcome the negative effects of high jitter, real-time audio/video clients typically have a buffer which buffers events up to 200 milliseconds and then proceeds to release them. In order to reduce the effect of high jitters in large distributed networked environment we provide a buffer whose size can be customized based on an entity's needs.

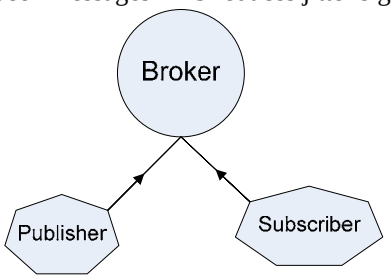
The Buffering Service within NaradaBrokering stores messages and releases them after sorting them according to their timestamps. The design of the buffering service has incorporated four configurable parameters pertaining to the release of time-stamped messages. The first criterion is the number of messages in the buffer maintained by the buffering service. If the number of messages reaches the maximum number of entries, it starts to release the time-ordered messages. The second criterion is the total size of the messages in the buffer. This along with the first criterion enables us to circumvent buffer overflows. The third criterion corresponds to the time spent by messages within the buffer. In some cases, the rate of messages arriving at an entity may be too slow and this may cause longer and unwanted delays within the buffer. The time-duration factor makes sure that the messages are released after a maximum specified duration if the first two criteria are not met. The final criterion is the release factor of the buffer. This typically has a value between 0.5 and 1.0. When any of the release criteria is met, it releases at least `release_factor X total_bufer_size` messages.

3.4 Time differential service

In collaborative systems simply receiving messages in time-order may not be enough. An entity may also place constraints on the maximum jitter that it is willing to tolerate. The Time Differential Service (TDS) provides two very important functions. First, it reduces the jitter in messages caused by the network. Second, it releases messages while preserving the time spacing between consecutive messages. Preserving time spacing between messages is not an easy task primarily because most operating systems do not provide strict real-time capabilities. Depending on the operating system, the scheduling of processes and threads does not necessarily guarantee the CPU for that process or thread after a specified interval. For example, using Java on the Windows operating system, user-level threads can obtain the CPU back only after 10 milliseconds. Based on the scheduling configuration of Linux operating system this duration can vary from 1 millisecond to 10 milliseconds or more.

One of the main reasons that TDS uses threads rather than traditional polling to release events in the queue is to avoid high CPU utilizations. In the case of polling, in order to release events in the queue their timestamps should be checked very frequently. This can lead to very high CPU utilizations. Furthermore, since rate at which events are generated is not constant: the time spacing between consecutive events vary. Using threads ensures that CPU utilizations are significantly lower. The reason that we have multiple threads instead of one thread to release the events in the queue is due to issues related to the underlying programming language (Java) and the operating system. For e.g. on Linux (Fedora 2), in order to check the timestamps every millisecond, we need to use at least three inter-leaving threads since each thread wakes up after a minimum of 3 milliseconds. On Windows, this value is 10 milliseconds; this high value may not be able to address jitter reduction adequately. .

TDS spawns five threads to process messages released by the buffering service. Note that TDS itself maintains another buffer for processing. Each thread is initiated one after another with a specified time difference between consecutive initiations. Each thread sleeps for a specified time-slice. By interleaving the durations at which these threads wake-up TDS can operate on the buffer at finer intervals while ensuring that CPU utilizations are low. The time-slice interval for individual threads impacts CPU utilization. We have observed that if the time interval between threads is 1 millisecond the CPU utilization stays around 5~6%, when this interval is decreased to 10 microseconds, it can reach about 20~25% on a Linux machine (1.5 GHz CPU 512 MB RAM). When a thread wakes up it checks to see if any messages need to be released, and does so if needed. It does so by comparing the message's timestamp, the local clock obtained from the high resolution timer and the time at which the last message was released. By preserving the time-spacing between messages TDS reduces jitter significantly.



Experiment 1:
Broker at Cardiff (Red Hat Linux 7.1 2.96-79, Pentium III , I GHz, 1.5GB RAM, JVM 1.4.1),

Clients at Indiana: Red Hat 3.4.2-6.fc3 Intel(R) Pentium(R) 4 CPU 1.5 GHz, 1 GB, JVM 1.4.2_07

Experiment 2:
Broker at Indiana (Red Hat Linux 3.2.3-6, Intel(R) Xeon(TM) CPU 2.40GHz, 2068240 kB, JVM:1.4.2_03),

Clients at Indiana: Red Hat 3.4.2-6.fc3 Intel(R) Pentium(R) 4 CPU 1.5 GHz, 1 GB, JVM 1.4.2_07

Figure 15: Experimental setup for TDS measurements

Figure 15 depicts the experimental setup for our TDS related measurements. The transmitter (publisher) captures the input-video stream from a camera and publishes them using NaradaBrokering messages, which are time-stamped appropriately. In the reported results, we ignored the first few messages that resulted in spikes due to media-player initializations.

Figure 16 contrasts the jitters resulting from the experimental setup (1) involving machines at Cardiff and Indiana; the graph compares the jitters in the Input to the buffering service and the Output of the TDS. Please note that in the absence of the buffering service and TDS at a client, the jitters experienced at that node would be similar to that corresponding to the input of the buffering service.

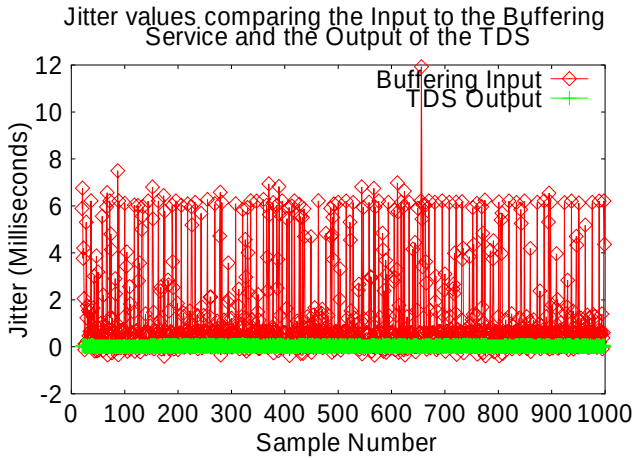


Figure 16: Jitter values comparing Buffering Service Input and TDS Output (Trans-Atlantic)

Figure 17 depicts only the jitters as a result of TDS for this experimental setup. The results demonstrate the significant reduction in jitter as a result of deploying the TDS.

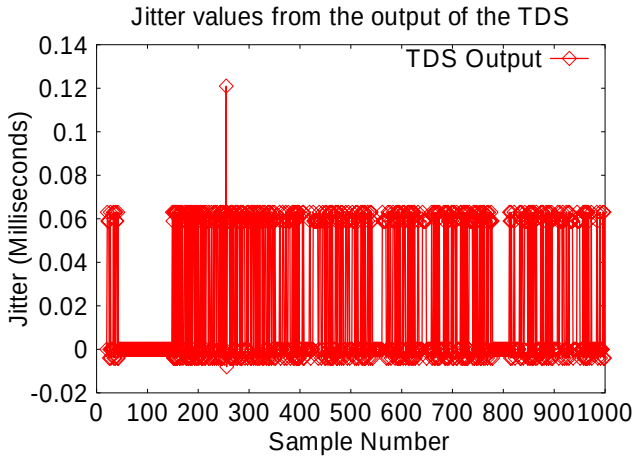


Figure 17: Jitter values from the TDS Output (Trans-Atlantic)

We also performed measurements in the same experimental setup by varying the time-slice intervals associated with the threads spawned by the TDS. Here we

report results (Figure 18) from our measurements for intervals of 1 millisecond and 100 microseconds. The results demonstrate that reducing the time intervals also reduces the jitter in the messages that are output by the TDS.

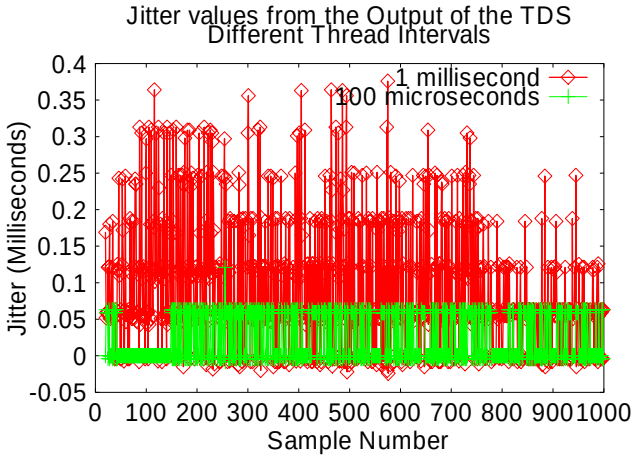


Figure 18: Jitter values from the TDS output for different thread intervals (Trans-Atlantic)

We have also performed measurements within a local area network to profile the performance of TDS, the results reported here correspond to the experimental setup (2) depicted in Figure 15. Figure 19 contrasts the jitters in the input to the buffering service and the output of TDS. Once again, the results demonstrate jitter reduction even in cluster settings.

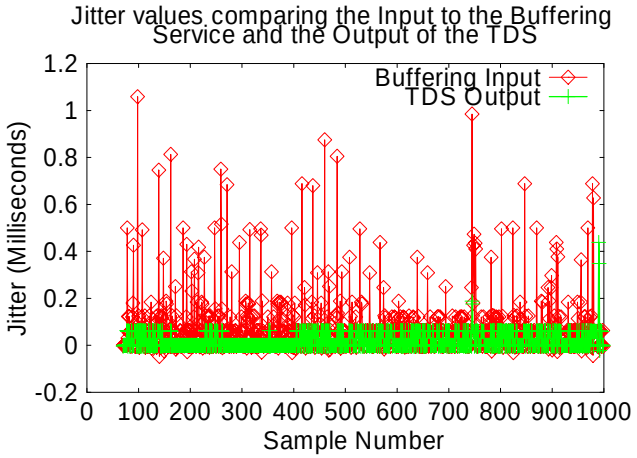


Figure 19: Jitter values comparing Buffering Service Input and TDS Output (Cluster setting)

In our last experiment we investigated if TDS would be able to space messages accurately if they were time stamped a few hundred microseconds apart. Here we generated messages that were spaced at intervals of 500 microseconds. Figure 20 depicts these results; the results demonstrate that TDS can be deployed in settings where messages are spaced a few hundred microseconds apart.

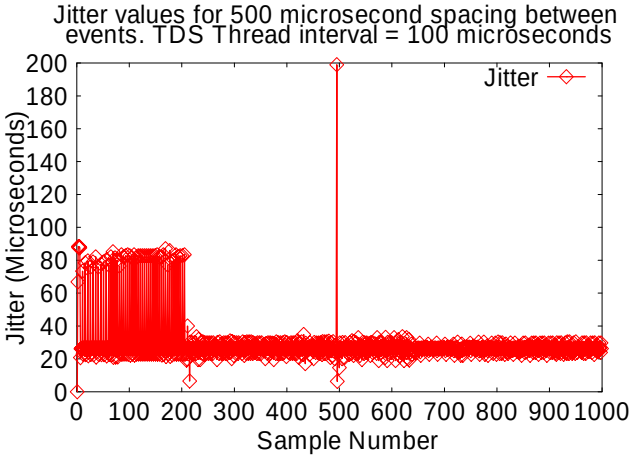


Figure 20: Jitter values for spacing messages with a fixed 500 microsecond spacing between them

3.5 Other NaradaBrokering services

NaradaBrokering also provides services to ensure reliable delivery of messages. This reliable scheme extends naturally to support replay services, ordered delivery and exactly-once delivery of messages. Additional information regarding these services can be found in Ref [4]. NaradaBrokering also includes services for monitoring the performance of individual links. It is also our contention that the NaradaBrokering security scheme [7] can be naturally extended to support secure interactions between entities.

To deal with messages with large payloads, NaradaBrokering provides services for compressing and decompressing these payloads. Additionally there is also a fragmentation service which fragments large file-based payloads into smaller ones. A coalescing service then merges these fragments into the large file at the receiver side. This capability in tandem with the reliable delivery service was used to augment GridFTP to provide reliable delivery of large files across failures and prolonged disconnects. The recoveries and retransmissions involved in this application are very precise. Additional details can be found in Ref [5]. Here, we had a proxy collocated with the GridFTP client and the GridFTP server. This proxy, a NaradaBrokering entity, utilizes NaradaBrokering's fragmentation service to fragment large payloads (> 1 GB) into smaller fragments and publish fragmented messages. Upon reliable delivery at the server-proxy, NaradaBrokering reconstructs the original payload from the fragments and delivers it to the GridFTP server.

4. Related Work

Examples of systems based on the generalized publish/subscribe paradigm include Elvin [10], Sienna [11] and Gryphon [12]. Elvin utilizes quench expressions to suppress producers from sending notifications while there are no consumers. This, however, entails each producer to be aware of all the consumers and their subscriptions. Sienna assembles patterns of notifications as close as possible to the publishers, while multicasting notifications as close as possible to the subscribers. Sienna however does not include support for ordering of notifications and expects the application to resolve it. In Gryphon each broker maintains a list of all subscriptions within the system in a parallel search tree (PST). The PST is annotated with a trit vector encoding link routing information. These annotations are then used at matching time by a server to determine which of its neighbors should receive that event. The Event Service [13] approach adopted by the OMG is one of establishing channels and subsequently registering suppliers and consumers to the event channels. The approach can entail clients (consumers) to be aware of a large number of event channels.

Another area of interest is peer-to-peer (P2P) systems, these include systems that are based on unstructured overlay networks (e.g. the old JXTA [14] model) and those that are based on structured overlay networks that employ Distributed Hash Tables (DHTs). DHTs have been quite popular in several P2P systems. Here each data object is associated with a key. A lookup service to locate this object returns the IP-address of the node hosting this object. Similar to a traditional hashtable data structure, other operations supported in the DHT include *put* and *get*. In DHT-based P2P overlay networks the nodes are organized based on the content that they possess. Here DHTs are used to locate, distribute, retrieve and manage data in these settings. This scheme provides bounded lookup times. Here examples of such systems include the current JXTA [14] system, Pastry [15]. FLAPPS [16, 17] provides a generalized infrastructure for peer network design. Here peers are organized into a peer network comprised of overlapping peer groups with transit peers efficient routing requests. It should be noted that though FLAPPS is not DHT based it can indeed be used to support DHT-style systems, Resilient Overlay Networks (RON) style networks and traditional peer networks. Unlike DHT systems with consistent hashing schemes data elements in Squid attempts to preserve locality and use the dimension-reducing mapping called Space Filling Curves (SFC). By preserving locality in the DHT index space Squid supports complex queries using partial keywords, wildcards and ranges.

A comprehensive discussion of the architectural similarities, differences, strengths and weaknesses of these systems vis-à-vis capabilities available within the NaradaBrokering substrate can be found in Ref [3]. Examples of collaborative infrastructures include JSDT [18] from Sun Microsystems. JSDT provides the basic abstractions of a session and also supports full-duplex multi-point connections between entities. Additionally, JSDT provides a token-based distributed synchronization mechanism to facilitate access to shared resources.

Approaches to synchronizing clocks in distributed systems include efforts such as Lamport's logical clocks [19], vector clocks [20] and Cristian's algorithm [21]. Lamport's logical clocks guarantees the order of events among themselves. They do not need to run at a constant rate, but they must increase monotonically. Using Lamport timestamps, Lamport synchronizes logical clocks by defining a relation called "happens-before". Vector clocks have been introduced because Lamport timestamps cannot capture causality. But a major drawback is that vector clocks add a vector timestamp, whose size is linear with the number of processes, onto each message in order to capture causality. Vector clocks thus do not scale well in large settings. In Cristian's algorithm, all of the machines in the system synchronize their clocks with a time server. Here each machine issues a request to the time server to retrieve the current time. The time server then responds to this message including its current time as fast as it can. Time server in Cristian's algorithm is passive. In the Berkeley algorithm [22], time server is active and polls every machine periodically. It then computes the average time based on the times received from the other machines and notifies them that they should adjust their time to the recently computed time.

5. Conclusions & Future Work

In this paper we have outlined the support for collaboration within the NaradaBrokering substrate which in turn provides for a richer collaborative experience. The various services related to ordering and delivery within the system can be leveraged by systems that mandate high-performance collaboration. Our experimental results demonstrate that the substrate can indeed be used in scenarios where performance and scalability requirements are stringent.

More recently we have included support for WS specification related to notifications and reliable delivery viz. WS-Eventing [23] and WS-ReliableMessaging [24]. This in turn will enable the development of collaborative

systems that are based on the emerging Web Services stack. The final version of this paper, if accepted, will incorporate performance measurements for the specifications.

References

[1] The NaradaBrokering Project at the Community Grids Lab: http://www.naradabrokering.org

[2] Shrideep Pallickara and Geoffrey Fox. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003.

[3] Shrideep Pallickara and Geoffrey Fox. On the Matching Of Events in Distributed Brokering Systems. Proceedings of IEEE ITCC Conference on Information Technology. April 2004. Volume II pp 68-76.

[4] Shrideep Pallickara and Geoffrey Fox. A Scheme for Reliable Delivery of Events in Distributed Middleware Systems. Proceedings of the IEEE International Conference on Autonomic Computing. pp 328-329. 2004.

[5] G. Fox, S. Lim, S. Pallickara and M. Pierce. Message-Based Cellular Peer-to-Peer Grids: Foundations for Secure Federation and Autonomic Services. Journal of Future Generation Computer Systems. Volume 21, Issue 3, pp 401-415 (1 March 2005). Published by Elseiver.

[6] Geoffrey Fox, Shrideep Pallickara and Savas Parastatidis. Towards Flexible Messaging for SOAP Based Services. Proceedings of the IEEE/ACM Supercomputing Conference 2004. Pittsburgh, PA.

[7] Shrideep Pallickara et al. A Security Framework for Distributed Brokering Systems. Available from http://www.naradabrokering.org

[8] RTP: A Transport Protocol for Real-Time Applications (IETF RFC 1889) http://www.ietf.org/rfc/rfc1889.txt.

[9] D.L. Mills. Network Time Protocol (Version 3). Specification, Implementation and Analysis. Internet RFC 1305. (March 1992)

[10] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe noti.cation service with quenching. In Proceedings AUUG97, pages 243–255, Canberra, Australia, September 1997.

[11] Antonio Carzaniga, et al Achieving scalability and expressiveness in an internet-scale event notification service. In Proceedings of the 19th ACM Symposium on Principles of Distributed Computing, pages 219–227 2000.

[12] G. Banavar et al. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In Proceedings of the IEEE International Conference on Distributed Computing Systems, Austin, Texas, May 1999.

[13] The Object Management Group (OMG). OMG's CORBA Event Service. Available from http://www.omg.org/

[14] Sun Microsystems. The JXTA Project and Peer-to-Peer Technology http://www.jxta.org

[15] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. Proceedings of Middleware 2001.

[16] B. Scott Michel, Peter L. Reiher: Peer-through-Peer Communication for Information Logistics. GI Jahrestagung (1) 2001: 248-256

[17] B. Michel and P. Reiher. Peer-to-Peer Internetworking. In OPENSIG, September 2001.

[18] Java Shared Data Toolkit (JSDT). http://java.sun.com/products/java-media/jsdt/index.jsp

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, vol. 21, no. 7 pp. 558-565, July 1978

[20] C.J. Fidge. Logical Time in Distributed Computing Systems. IEEE Computer. vol.24, no.8, pp.28–33, 1991.

[21] F. Cristian. Probabilistic clock synchronization. Distributed Computing. 3:146–158, 1989.

[22] R. Gusella, S. Zatti. The accuracy of clock synchronization achieved by TEMPO in Berkeley Unix 4.3BSD. In IEEE Transactions on Software Engineering. Vol. 15, pp. 847-853

[23] Web Services Eventing. Microsoft, IBM.. http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf

[24] Web Services Reliable Messaging Protocol (WS-ReliableMessaging) ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200403.pdf