RZ 3685 (# 99695) 03/26/2007
Electrical Engineering 8 pages

Research Report

Tempo
A Simple Time-Sensitive Messaging System

Daniel Bauer, Luis Garcés-Erice, Sean Rooney

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
e-mail: (dnb, lga, sro)@zurich.ibm.com

Tempo
A Simple Time-Sensitive Messaging System

Daniel Bauer, Luis Garcés-Erice, Sean Rooney
IBM Research, Zurich Research Laboratory Säumerstrasse 4.
8803 Rüschlikon, Switzerland

***Abstract*— Tempo is a messaging system designed for time-sensitive applications that require low-latency and high data rates. In Tempo, unlike existing real-time messaging systems, most of the policy is delegated by the messaging layer to the application. It is the application's responsibility to react to information supplied by the messaging layer in order to achieve some performance objective; this makes Tempo suitable for use in environments which must reconfigure often and hence in which the normal Real-Time techniques such as Worst Case Execution Time analysis would be impossible or prohibitively expensive.**

I. Introduction

As the physical world becomes increasingly connected to computer networks through intelligent sensors and actuators there are more applications that require knowledge of the time that it takes to achieve various tasks. Doing so enhances existing applications and enables entirely new ones as underlying systems can be monitored and changed at granularities of time that were previously infeasible. Examples, of such applications are found in telematics, industrial automation, health care.

Messaging systems are attractive in such an environment as they promote a data-centric form of communication in which the interests of entities determines the associations between them. This allows an entity at the application level to be oblivious as to which other entities it is communicating with, permitting a looser coupling between entities than in an address-centric approach. An appropriate messaging system should be simple enough that it can be added to low-end devices, be self managing, scale to large number of end-points and support high data rates with bounded delay.

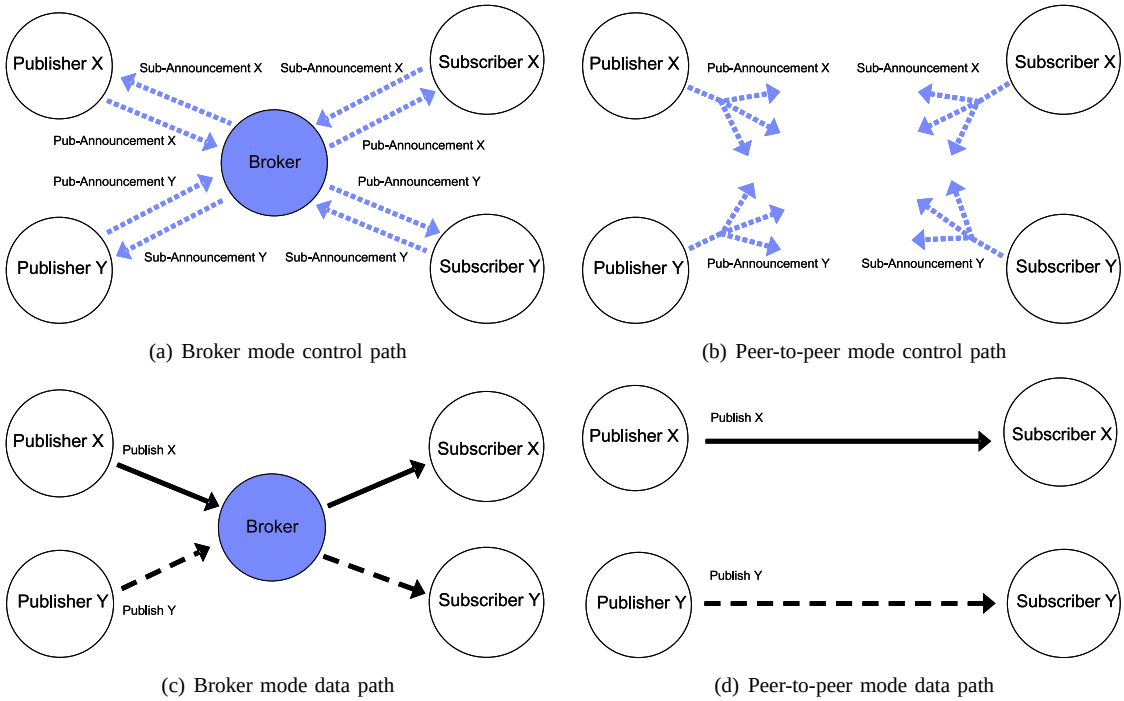
We describe a messaging system called Tempo that meets the design goals that we have outlined. Tempo is a 'bare bones' messaging system allowing it to scale to high data rates and high numbers of end-points with low latency. Within Tempo, relative guarantees can be associated with the delivery of messages on different topics, allowing applications to privilege the delivery of certain message types over others. This is achieved through a design which allows resources, e.g. CPU, to be allocated to distinct topics. Tempo monitors and makes available information about the performance of the messaging layer to applications allowing them to dynamically instrument application-specific policies which can adapt to changing circumstances. Tempo is implemented using Real-Time Java (RTSJ) [1] enabled with the Metronome Real-Time garbage collector [2].

II. Related Work

Existing commercial messaging system have been designed with transactional applications in mind. In such systems, for example for stock trading, reliability and non-repudiation are the most important design goals, in consequence they are inappropriate for the types of application we have in mind.

Some attempts have been made to design Real-Time messaging systems: CORBA [3] offers an event service that supports a publisher/subscriber event mechanism over the basic CORBA/IIOP. RT-CORBA can be used to tune an upper-bound in the time it takes to achieve a distributed activity through the allocation of resources, e.g. thread priorities, across a distributed environment. The real-time CORBA Event Service [4], an extension to CORBA's event service, takes advantage of real-time CORBA's capabilities as well as simplifying the communication protocol. CORBA does not achieve our other design goals; it is a general purpose Distributed Processing Environment not just a messaging system, meaning that end-systems have to support the complete set of APIs required by the Object Management Group (OMG). This is not favorable to low-end devices. In addition, the requirement that standard CORBA protocols are used within the messaging service introduces unnecessary overhead. Furthermore, CORBA's many layers of software make it difficult to analyze the performance of the resulting system. The need for an explicit event server supporting the publish/subscribe application necessitates an extra hop across which each message must be carried even if the publisher and subscriber are in the same broadcast or multicast domain. Finally, CORBA's use of explicit bindings whereby publishers and subscribers explicitly connect and unconnect from the event server make it unsuitable for environments where devices often fail and in which connectivity is lost and reestablished.

Unlike CORBA, Data Distribution Service (DDS) [5] is explicitly a publish/subscribe infrastructure. It does not define a wire protocol but simply a set of APIs defined using CORBA IDL. DDS implementations consequently do not inter-operate. DDS offers some support for resource allocation. A publisher can announce a rate that it wishes to publish at and subscribers at a rate at which they wish to receive and DDS ensures their consistency. An end-to-end latency requirement may be associated with a message but this is only a hint to the application and DDS ignores it. DDS has quite a rich set of operations describing different semantics for message delivery. In particular, DDS allows a data item to be associated with



(a) Broker mode control path

(b) Peer-to-peer mode control path

(c) Broker mode data path

(d) Peer-to-peer mode data path

Fig. 1. Path of control and data messages in Tempo

a key, thereby giving it an identity. Multiple data items with the same key may exist within DDS simultaneously and the delivery semantics determine whether old ones overwrite new ones or are retained. Data items can be retained until explicitly removed by subscribers; data items can be expired after a given time period.

III. Publish/Subscribe Protocol

In a publish/subscribe system publications on a topic are sent to all subscribers to that topic. The publish/subscribe abstraction can be supported using a dedicated broker whose role is to dispatch published messages to the appropriate subscribers. Alternatively the topic/subscribers association can be maintained at the publishers, such that publishers send directly to subscribers. The peer-to-peer case incurs less latency as an in-direction through a broker is avoided however it is only appropriate for small numbers of publishers and subscribers due to the large amount of state that needs to be maintained at the end-points.

Tempo supports both modes of operation within the same publish/subscribe protocol allowing the trade-off between latency and scalability to be determined by the user[1]. A publisher periodically announces its ability to send messages on a named topic and a subscriber periodically announces its readiness to receive messages on a named topic. Publishers and subscribers communicate with one another across an IP addressable channel. The channel can be a broker or simply the broadcast or multicast domain. These two cases are transparent to the publishers and subscribers and are configurable simply through the address, i.e. in one case it is the broker's address, in the other it is the broadcast or multicast address.

Publishers maintain the list of subscribers to their topic and send publications to that list of subscribers. If after some time a subscription announcement has not been received from a subscriber, the publisher removes the associated state about that subscriber and will not send any further messages to it. In the broker case, the broker acts as a proxy appearing as a publisher to subscribers and a subscriber to publishers. It aggregates information between multiple announcement messages meaning that although there may be many publishers and subscribers on a given topic, from an end-point's point of view there is only ever one i.e. the broker.

A publisher sends in its announcement the rate at which it wishes to publish on the topic and subscribers send the rate at which they wish to receive. These messages are sent periodically and the rates can vary over time. The rate announced by a publisher or subscriber running on an end-point are used locally within the scheduling mechanism to allocates shares to the corresponding topics as described in Section IV. Note that these rates are not the actual rate at which data is being sent but are descriptions of the intended sending and receiving rates. Figure 1 compares the data and control paths of messages in the broker and peer-to-peer mode.

Tempo runs over both UDP and TCP. The messaging system itself never reorders or discards messages meaning that if TCP is used messages are delivered reliably. If UDP is used, messages may be lost in the network layer. The choice of transport protocol changes the system characteristics that we can optimize, with UDP delay is minimized but arbitrary rates of loss are possible, with TCP there is no loss but arbitrarily long delays are possible.

Each end-point that is acting as a publisher on a topic

[1]Hybrid modes of operations are also possible in which a given message may be sent to some subscribers directly while others receive it through a broker but these are not described here due to lack of space.

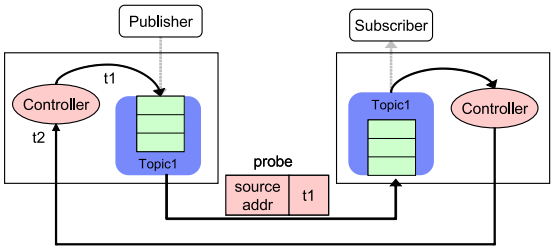


Fig. 2. Sending probes to estimate end-to-end delay

periodically transmits a probe message. The probe is carried through the normal data path to all subscribers. Contained in the probe is the time at which it was sent as well as the source address of the sender. The probe is echoed back at the subscriber to the publisher allowing the publisher to estimate the current end-to-end delay on that topic between the publisher and all subscribers. The return path of the probe does not go though the messaging system meaning that it is faster than the forward path allowing a publishing application to use the round trip time as a conservative upper bound for the forward path at the current point in time.

Figure 2 shows the general schema of the means for estimating end-to-end delay. The controller places a probe in the messaging system at time t_1. The probe is carried through the messaging system to the subscribers, but instead of being delivered to the applications it is echoed back. The publishing controller, receiving the echoed probe at time t_2 calculates the round-trip as $t_2 - t_1$ this serves as a conservative upper-bound for the end-to-end delay. The extent to which the round trip time overestimates the end-to-end delay depends on the relative amount of time spent in the network compared to the messaging system. When the network is unloaded and the propagation delay is short then we can expect the calculated time to be a reasonable approximation, when the network is congested or the propagation delay is large the round trip may be twice as large as the actual end-to-end delay.

Each application participating in a topic receives periodic reports about the performance of the publishers and subscribers to that topic. Applications can then make use of this information in application-specific ways. For example, a publishing application may adapt its sending rate to match the lowest subscription rate. Section V describes this in more detail.

IV. The Messaging Engine

Tempo has been designed to support applications that require a messaging service with timeliness guarantees, where timeliness means both guaranteed throughput in terms of messages per second and bounded latency between publishers and subscribers. Timeliness is achieved by carefully allocating the CPU. This happens on multiple levels, as shown in Figure 3. At the coarsest level, shown at the bottom of the Figure, the CPU is divided between the Tempo middle-ware and other applications. It is the task of the operating system to schedule threads of individual applications. Tempo runs in a real-time Java environment that provides a fixed-priority scheduler as well as periodic threads. Tempo assumes a rate monotonic [6]

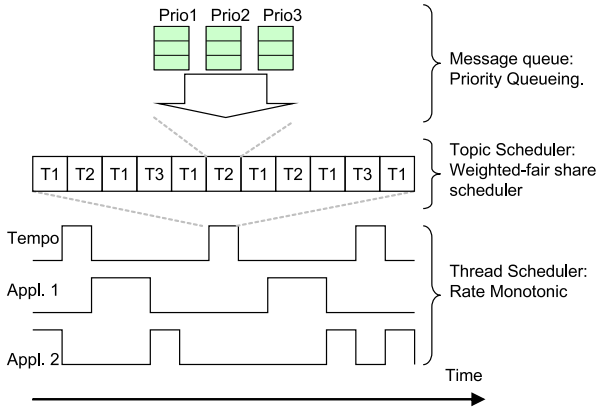


Fig. 3. Scheduling Hierarchy

scheduling discipline that is used by all threads. Inside the Tempo middle-ware, a topic scheduler decides how the CPU is divided among different topics. Here, a weighted-fair share scheduler is used that serves each topic according to its share, where a share corresponds to a message rate. Finally, on the level of individual messages, priorities are used to distinguish messages within the same topic. High priority messages skip low priority ones and are thus forwarded with a lower delay.

A. Thread Scheduling

The operating environment provides a thread scheduler that controls access to the CPU. RTSJ mandates a fixed-priority scheduler that is available on most, if not all, real-time operating systems. This scheduler requires that applications cooperate, otherwise an application constantly running at high priority starves other applications. While other thread schedulers such as proportional fair share schedulers, for example [7], have been the subject of active research, few commercial implementations exist. Tempo's design builds on the fixed-priority scheduler. This scheduler does not use time-slicing. A running thread will continue to use the CPU until a thread with higher priority becomes ready to run or until it blocks on I/O or yields.

Tempo uses two different approaches for thread scheduling that can be described as time-driven and load-driven. In order to simplify the discussion in this Section, it is assumed that Tempo and the other applications are single threaded and the term application or thread is used synonymously. Section IV-D describes Tempo's multi-threaded implementation.

In the time-driven approach, all applications are periodic and run for a bounded time during each period. The scheduling discipline is the rate monotonic algorithm [6] that assigns a fixed priority to each thread, threads with lower period have higher priority. The rate monotonic algorithm can be implemented in an RTSJ environment, but it requires that applications cooperate, i.e. they must yield voluntarily after they've used up their CPU budget. In addition, stopped applications must be released periodically, a feature that is provided by RTSJ's thread implementation. While the approach based on rate-monotonic requires strictly periodic threads, messages do not arrive periodically. Tempo handles the asynchronous

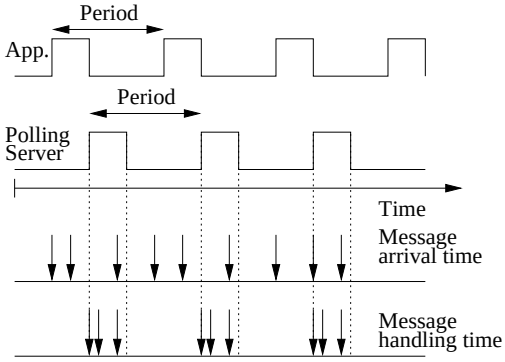


Fig. 4. Polling Server

arrival of messages by implementing a polling server [8]. A polling server is a periodic thread and therefore is eligible for being scheduled by the rate-monotonic scheduling discipline. Using the polling server approach, Tempo is executed periodically and handles pending messages. Messages that arrive when Tempo is not active, are buffered, for example in the OS's network buffers. Figure 4 shows a schema of the polling server approach.

The period of the messaging system affects the delay of the messages. In the time-driven approach, the lowest bound on delay can not be smaller than the period. This approach is useful when a high level of control is required and message latency of the same order as the scheduling period can be tolerated. While the scheduling period is a configurable parameter, typical values are in the range of several milliseconds.

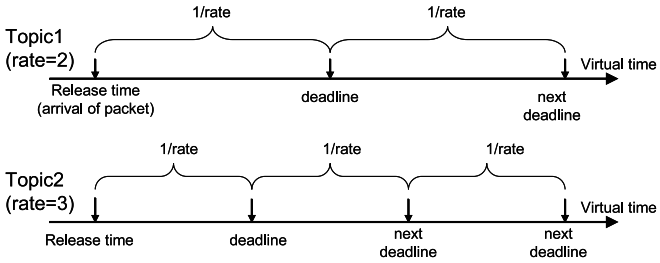
In the load-driven approach Tempo is assigned the highest priority in the system. It runs not periodically, but whenever there is work to do. The CPU time that is used depends on the load and it is possible that with this approach, Tempo uses all of the available CPU. The load-driven approach is useful for event-driven applications or for applications in which message delay is of primary concern and the load is small.

B. Topic Scheduling

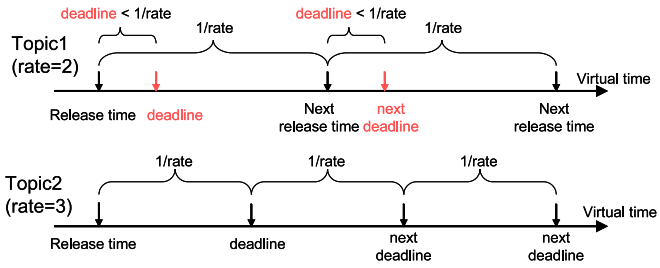
The second level of CPU resource control decides how the CPU is used within the Tempo middle-ware. The schedulable objects at this level are not threads but topics. The topic scheduler assigns CPU shares to topics to forwards messages. The scheduling unit is not a time unit but the task of forwarding a single message to all its destinations.

The topic scheduler uses weighted-fair-sharing with optional deadline constraints. Fair share scheduling is implemented using virtual time and the earliest-deadline-first selection discipline. The topic scheduler is made independent of the thread scheduling by using virtual time instead of real-time. The topic scheduler partitions the CPU slice allocated to Tempo by the thread scheduler according to the topic shares. Excess resources are divided among the active topics according to their shares, thus the topic scheduler is work conserving, i.e. it will work when there is work to do.

For each topic where at least one waiting message exists, a deadline for handling it is computed. Each time a message is taken from a topic, the virtual time is advanced to the next deadline. The share that a topic is allocated can also be seen as a message rate, measured in virtual time. This is shown in Figure 5(a). Here, two topics exists. Topic T_1 is given a rate of



(a) Mapping Rates to Shares



(b) Prioritizing

Fig. 5. Assigning Deadlines to Topics

2 and Topic T_2 is given a rate of 3. In the basic configuration, the reciprocal values of the rates are used for the deadlines. In this particular example, multiple schedules exist, for example $T_2, T_1, T_2, T_1, T_2, T_2, T_1, \cdots$ or $T_2, T_1, T_2, T_2, T_1, T_2, T_1, \cdots$. In order to control message latency, the topic scheduler allows the setting of a deadline that is independent of the rate, as long as this deadline is smaller than the reciprocal value of the rate. This is shown in Figure 5(b). In this case, a unique schedule exists: $T_1, T_2, T_1, T_2, T_2, T_1, T_2, \cdots$ Note that reducing the deadline does not change the share of the topic.

Setting the deadline to less than $1/\text{rate}$ is useful for topics that have a low message rate and thus a small CPU share but require low latency. For example, a topic might be used for the transmission of alert events that happen very rarely but that must be forwarded as quickly as possible. The deadline of the topic is computed when a message arrives. Due to the small share, the default deadline will be rather large and the message will experience a high delay. Adding the additional deadline decouples rate from latency and allows the configuration of topics for low rate, low latency messaging.

C. Message Prioritization

Most applications require that messages within a topic are delivered in order. However, there are applications that require that certain messages are forwarded out of order, for example messages that represent alarm events. These messages have to be delivered with the smallest possible delay. Tempo supports the prioritization of messages within a topic such that high priority messages can skip ahead of low priority messages within a topic. Messages of the same priority are forwarded in FIFO order.

D. Implementation

Figure 6 shows the implementation of the data path of Tempo in peer-to-peer mode, publishers on the left and subscribers on the right.

Topics are the central data structure. A topic consists of an input queue that receives messages either from the application when used by a publisher or from the network when used by a subscriber. Furthermore, each topic contains a set of destinations to which messages are delivered, i.e. the set of subscribers. Finally, topics contain state information that is used by the topic scheduler.

The implementation of the data path in the broker is shown in Figure 7. In the case of the broker, the transport layer delivers messages into the input queues, from where they are selected by the topic scheduler and forwarded out through the transport layer again. The software components used by the broker are the same as those used in the endpoints.

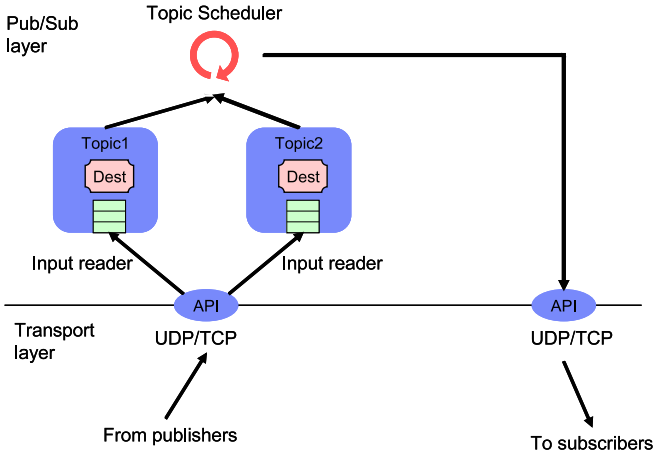


Fig. 7. Data Path in Broker Mode

Tempo is designed for real-time Java on real-time Linux. This environment supports two thread types: real-time threads and standard threads. The standard threads are scheduled using a time-sharing approach, where the priority of threads is reduced the longer they run. Real-time threads are scheduled using a fixed-priority scheduler. A real-time thread runs until it yields or until a thread with a higher priority is ready to run. Real-time priorities are always higher than the priorities of normal threads.

The garbage collector is an integral part of each Java environment and it effects the runtime behavior of threads. Although RTSJ supports threads that are not stopped by garbage collection, those threads are not allowed to allocate memory on the heap and are therefore not suitable for Tempo. Instead, Tempo runs on an RTSJ implementation that is equipped with a real-time collector called Metronome. Metronome is an incremental collector that runs periodically when there is work to do. The current implementation [9] of Metronome pauses threads for typically $500\mu s$, with maximum pause times of about $3ms$. The maximum CPU utilization of Metronome is configurable and a typical value is to not use more than 25% of the CPU during collector runs. The low pause times achieved by Metronome allows the delay of messaging in Tempo to be bounded to a few milliseconds, which is sufficient for a large range of applications.

Due to restrictions imposed by the Java runtime environment, Tempo is multi-threaded and uses two kinds of threads: Input threads that read messages from the network and a real-time thread for the topic scheduler. The Input threads are indirectly controlled by the topic scheduler and thread synchronization is done using the input queues. Input threads put messages into a wait-free read queue [1] that blocks when the queue is full, see also Figure 6. The queue is wait-free for reading and therefore the topic scheduler never blocks.

The polling server approach requires that all threads are periodic. The input threads, however, do not run periodically but whenever there is work to do. Tempo can still use the polling server approach with appropriate configuration. Since the input threads run aperiodically, they must run in the background when no other periodic thread is executed. Furthermore, the input threads must still run with the same periodicity as the topic scheduler. This is achieved by using the same period for all applications and leaving enough slack for the input threads to run, as shown in Figure 8

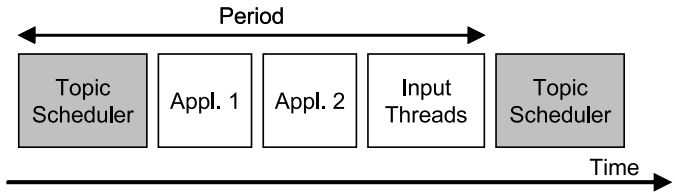


Fig. 8. Periodicity of Topic Scheduler and Input Threads

E. Analysis

Assume that if the messaging system ran by itself that an average service rate of μ would be achievable. Supposing that there are N topics such that S_i is the rate allocated to topic *i*, then the guaranteed service rate of the jth topic is given by:

$$\mu_j = \mu \frac{S_j}{\sum_{i=1}^{N} S_i}$$

For the sake of analysis we assume that the inter-arrivals and service times are exponentially distributed such that μ_j and λ_j are the average service and arrival rate for the topic *j*. From experiments we observe that for a given message size the service time is deterministic, but if the size of the messages is exponentially distributed we would expect a corresponding distribution for the service times. The *intensity* of the topic is then given by:

$$\rho_j = \mu_j / \lambda_j$$

The mean response time in the messaging system i.e. average time it takes to be forwarded, is then given by [10]:

$$E[\text{response time}] = \frac{1/\mu_j}{1-\rho_j}$$

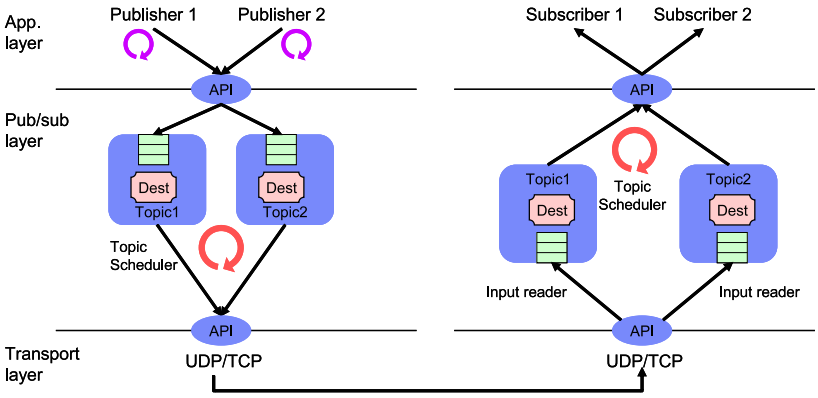


Fig. 6. Data Path in Peer-to-Peer Mode

and the q-percentile of the response time, i.e. the value at which q percent of the response time is expected to be inferior, is:

$$E[\text{response time}] \cdot ln[\frac{100}{100-q}]$$

For example, if $\mu_j = 10000$ messages/sec and $\lambda_j = 250$ messages/sec, then the average response time is 0.1 ms and 99.9% of all packets are treated within 0.7 ms.

In the time-driven approach the thread scheduling model means that the messaging thread can only run for time *T*, every period *P*. In effect it reduces the guaranteed service rate allocated to a topic by *T/P*. For a D/D/1 system with $\mu_j \cdot T/P > \lambda_j$ the expected time in the system is P/2 and the maximum is P. For a M/M/1 system there is some probability of a message remaining in the queue across multiple periods. This can occur if the number of messages that arrive in time P exceeds the number that can be serviced in time T. As the inter-arrival rate is exponentially distributed then the probability of k arrivals within the period P-T, i.e. the time Tempo does not run, is given by the Poisson distribution; the system during T can then be treated as a classic M/M/1 system with an initial queue length given by this distribution. The probability of no messages staying in the system more than one period P is the probability that all arrivals within time T are treated within time T and that there is enough idle time to remove the backlog k.

As explained in Section IV-B a topic i may have a deadline d_i in addition to a rate r_i. By definition, if d_i is defined then $d_i \leq 1/r_i$. Deadlines affect the order in which messages from different topics are serviced without changing the share allocated to each topic. Let $w_i = min(d_i, 1/r_i)$, and $Q = \{w_j \mid i \neq j, w_j \leq w_i\}$ then in the load-driven approach the average time a topic i will wait to be serviced is:

$$\frac{\sum_{w_k \ in \ Q} \lceil w_i/w_k \rceil}{\mu}$$

This must be less than $1/r_i$ for the system to be stable.

V. API

The Application Programming Interface (API) contains: the means for a subscriber to register a function to callback when a message arrives on a topic; the means by which a publisher can publish a message on that topic. The API also allows publishers and subscribers to change the rate and deadline associated with the corresponding topic. This allows applications to attempt to calibrate the system to achieve an acceptable performance. The application is periodically informed of the current performance of the system. In particular, publishers are informed of the end-to-end delay and the announced subscription rates on a topic and subscribers are informed of the loss-rate and the announced deadlines of publishers.

How applications make use of the above information is application-specific. A subscriber experiencing loss might use the API to reduce the rate for that topic. The publishing application receiving this information might then adapt to reduce its sending rate to the lowest announced subscription rate; by repeating this process a loss free rate from publishers to subscribers can be determined. A publisher with a target maximum end-to-end delay learns if the measured delay exceeds this and can reduce the deadline on the topic. All subscribers will learn of this change in deadline through the control messages and can change their deadlines as well. This process is continued until an acceptable end-to-end delay is achieved or it is clear that the delay is not achievable. The exact algorithm used, changes as a function of the needs of the application but only subscribers know the actual number of messages being achieved and only publishers know the actual end-to-end delay. Figure 9 shows information received at the publishers and subscribers and an example of the way in which they can respond.

VI. Results

In this section we report performance figures that have been obtained with Tempo on machines that are equipped with 3 Ghz dual Xeon processors, 2 GB of RAM running a real-time version of Linux and IBM's real-time Java environment. We provide a comparison with CORBA's real-time event service on the same setup, using the TAO implementation [4].

First we present latency measurements on a single publisher/subscriber pair. The transport mechanism is TCP. The publisher sends at a constant rate of 250 messages per second, each message of size 128 bytes. The experiment runs for 16 hours, resulting in a total of 14.4 million messages sent.

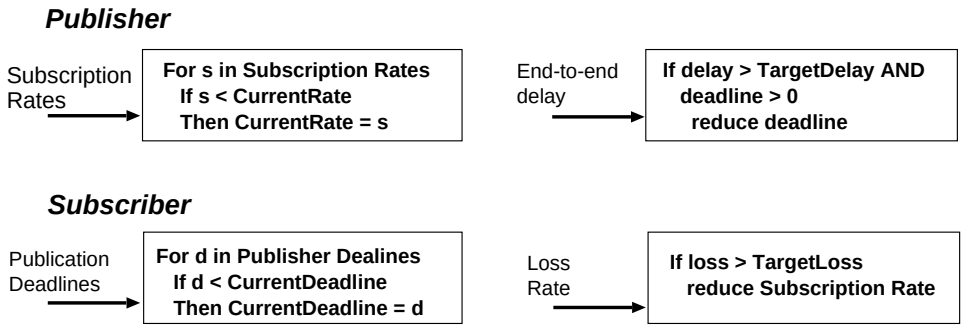


Fig. 9. Example of application-level adaptation

In a first experiment, Tempo is configured in peer-to-peer mode with the load-driven scheduling approach. The resulting latency distribution can be seen in Table I. While more than 95% of all messages are delayed less than $300\mu s$, there are a few messages that are delayed up to $10ms$.

Delay bound	Percentage of all Messages
Less than $300\mu s$	95.28022%
Less than $500\mu s$	99.98569%
Less than $1000\mu s$	99.99563%
Less than $2000\mu s$	99.99986%
Less than $10000\mu s$	100%
Average	$235\mu s$
Median	$220\mu s$

TABLE I
DELAY DISTRIBUTION TEMPO PEER-TO-PEER

In a second experiment, Tempo uses a broker. A single publisher/subscriber pair is executed on one machine and the broker on another, identical machine. The two machines are connected by a gigabit ethernet network. Table II shows the result. The additional delay of the broker as well as the network delay are visible and contribute to an increase of the latency of about $200\mu s$. Nevertheless, more than 99% of all messages experience a delay of less than $500\mu s$.

Delay bound	Percentage of all Messages
Less than $300\mu s$	0%
Less than $500\mu s$	99.92622%
Less than $1000\mu s$	99.98165%
Less than $2000\mu s$	99.99867%
Less than $10000\mu s$	100%
Average	$383\mu s$
Median	$380\mu s$

TABLE II
DELAY DISTRIBUTION TEMPO BROKER MODE

Next we show the end-to-end delay measured on the same set-up, but using CORBA's real-time event service. We place a subscriber and a publisher on one machine, while we install the real-time event service on another. The delay distribution is presented in Table III. This distribution shows a somewhat higher average and median latency compared to Tempo. The vast majority of all messages are delivered in less than $1ms$, although a very few messages get delay up to $14ms$.

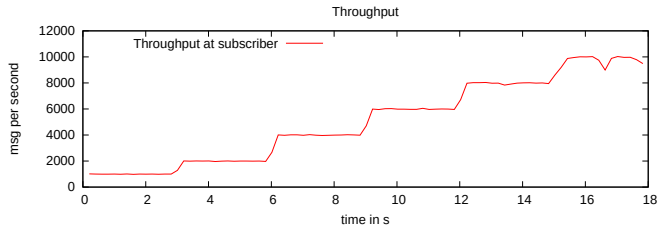
Delay bound	Percentage of all Messages
Less than $300\mu s$	0%
Less than $500\mu s$	3.38493%
Less than $1000\mu s$	99.99916%
Less than $2000\mu s$	99.99954%
Less than $10000\mu s$	99.99995%
Average	$526\mu s$
Median	$520\mu s$

TABLE III
DELAY DISTRIBUTION CORBA RT-EVENT SERVICE

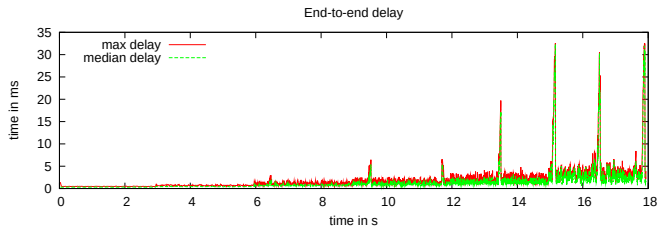
Figure 10(a) shows the trade-off between throughput and end-to-end delay. In the experiment, the sending rate was increased every 3 seconds until saturation was reached at 10,000 msgs/s. The corresponding end-to-end delay increases as the sending rate increases, becoming highly volatile at sending rates higher than 8000 msgs/s. This is due to the Metronome garbage collector running more frequently as the memory allocation rate increases. Each of the peaks in Figure 10(b) corresponds to a garbage collection. Garbage collection is visible in Figure 10(c) at positions where the free memory size suddenly increases. Metronome uses a configurable, bounded share of the CPU. Tempo's CPU consumption increases with the sending rate and therefore a garbage collection affects Tempo more strongly at high rates when few spare CPU cycles are available.

We are unable to give comparable figures for CORBA as the implementation seems to fail at rates higher than 4000 msgs/s. At this rate, CORBA showed an an average end-to-end delay of $950\mu s$, while Tempo achieved less than $750\mu s$.

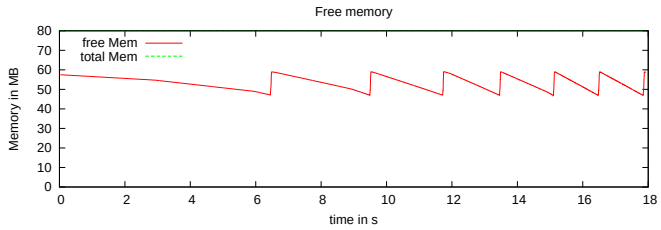
The isolation experiment tests how the end-to-end delay of a single publisher-subscriber pair is affected by another publisher-subscriber pair. The experiment features two clients, each running a publisher and a subscriber publishing on different topics, as well as a broker. Both publishers are allocated the same CPU share and initially publish 1000 msg/s. During a period of 10 seconds, the "misbehaving" publisher increases its rate to 7000 msg/s and then drops back to 1000 msg/s. The clients' share does not change. During the contention period, the broker operates close to its capacity. Figure 11 shows the delay of the well-behaving client. During the contention period, the experienced delay triples. However, in absolute values, it increases by only 3 ms. Complete isolation can be achieved by allocating resources to topics throughout the



(a) Throughput



(b) Delay



(c) Memory

Fig. 10. Achievable Throughput in Tempo

system. In Tempo, however, some resources are still shared: for example the input threads reading from the sockets compete for the CPU, the kernel's network stack does not provide resource sharing and neither does the network interface.

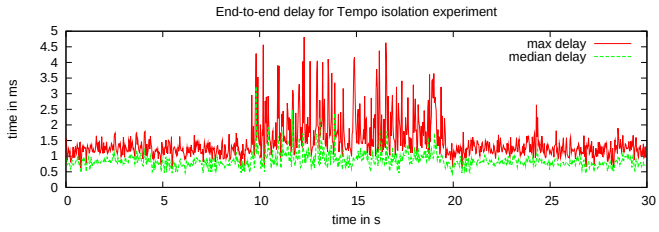


Fig. 11. End-to-end delay, well-behaving client

The picture looks different for the non-respecting client, as shown in Figure 12. During the contention period, the messages experience a delay that is several orders of magnitude higher than during normal operation. In addition, there are several peaks of short duration. These delay peaks are caused by garbage collection in the broker.

In conclusion, our real-time Java implementation achieves a comparable end-to-end delay compared to the real-time event service of CORBA. The achievable throughput of Tempo appears to be superior. Tempo allows the isolation of one topic from another through the allocation of CPU shares, no comparable means of resource allocation for topics exists in CORBA.

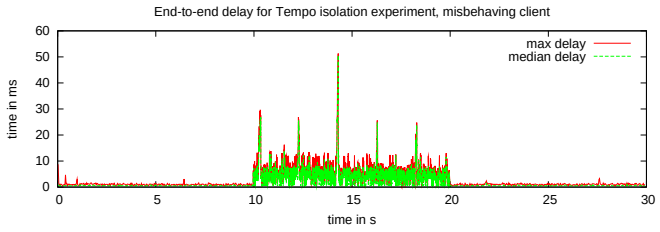


Fig. 12. End-to-end delay of Tempo, misbehaving client

VII. Conclusion

We have shown how time-sensitive applications can make use of a simple messaging system to achieve high-data rates and bounded delays without requiring extensive analysis. The messaging system achieves this by reporting aspects of the performance to applications and allowing them to calibrate the messaging system to achieve performance targets. We have described the protocols and algorithms the messaging systems uses and have shown that 250 messages/per second can be delivered over our test infrastructure where 99.995% messages are not delayed more than 1 millisecond and that our system supports up to 10,000 msgs/s before reaching saturation.

References

[1] P. Dibble *et al.*, "JSR 1: Real-time specification for java." `http://jcp.org/en/jsr/detail?id=1`, July 2006.

[2] D. F. Bacon, P. Cheng, and V. Rajan, "The metronome: A simpler approach to garbage collection in real-time systems," in *Workshop on Java Technologies for Real-Time and Embedded Systems* (R. Meersman and Z. Tari, eds.), vol. 2889 of *Lecture Notes in Computer Science*, pp. 466–478, Nov. 2003.

[3] OMG, *CORBA Notification Service Specification*. Object Management Group Publication, Aug. 2002.

[4] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time CORBA event service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), pp. 184–200, ACM, October 1997.

[5] OMG, *Data Distribution Service for Real-time Systems, v1.1*. Object Management Group Publication, May 2004.

[6] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *JACM*, vol. 20, no. 1, pp. 46–61, 1973.

[7] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "Surplus fair scheduling: A Proportional-Share CPU scheduling algorithm for symmetric multiprocessors," in *Proc. of the 4th Symposium on Operating Systems Design and Implementation*, (San Diego, CA), pp. 45–58, Oct. 2000.

[8] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers*, vol. 44, pp. 73–91, Jan. 1995.

[9] D. F. Bacon, P. Cheng, D. Grove, M. Hind, V. Rajan, E. Yahav, M. Hauswirth, C. Kirsch, D. Spoonhower, and M. T. Vechev, "High-level real-time programming in java," in *Proc. of the Fifth ACM International Conference on Embedded Software*, (Jersey City, New Jersey), Sept. 2005.

[10] R. Jain, *The Art of Computer Systems Performance Analysis*. Wiley Professional Computing, 1991.