Stheno, a Real-Time Fault-Tolerant P_2P Middleware Platform for Light-Train Systems[*]

Rolando Martins
EFACEC & CRACS & INESC-TEC
University of Porto
rolando.martins@efacec.com

Luís Lopes   Fernando Silva
CRACS & INESC-TEC
University of Porto
{lblopes,fds}@dcc.fc.up.pt

Priya Narasimhan
ECE Department
Carnegie Mellon University
priya@cs.cmu.edu

ABSTRACT

Large scale information systems, such as public information systems for light-train/metro networks, must be able to fulfill contractualized Service Level Agreements (SLAs) in terms of end-to-end latencies and jitter, even in the presence of faults. Failure to do so has potential legal and financial implications for the software developers. Current middleware solutions have a hard time coping with these demands due, fundamentally, to a lack of adequate, simultaneous, support for fault-tolerance (FT) and real-time (RT) tasks. In this paper we present Stheno, a general purpose peer-to-peer (P_2P) middleware system that builds on previous work from TAO and MEAD to provide: (a) configurable, transparent, FT support by taking advantage of the P_2P layer topology awareness to efficiently implement Common Of The Shelf (COTS) replication algorithms and replica management strategies, and; (b) kernel-level resource reservation integrated with well-known threading strategies based on priorities to provide more robust support for soft real-time tasks. An evaluation of the first (unoptimized) prototype for the middleware shows that Stheno is able to match and often greatly exceed the SLA agreements provided by our target system, the light-train/metro information system developed and maintained by EFACEC, and currently deployed at multiple cities in Europe and Brazil.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [design studies, fault tolerance, reliability, availability, and serviceability]; C.1 [**Processor Architectures**]: Parallel Architectures—*distributed architectures*; C.3 [**Special-Purpose and Application-Based Systems**]: [real-time and embedded systems]

Keywords

Peer-to-Peer, Middleware, Fault-Tolerance, Real-Time, Resource Reservation

1. INTRODUCTION

At EFACEC[1] we have to handle a multitude of application domains, including: information systems used to manage public, high-speed, transportation networks; automated power management systems that handle smart grids, and; power supply systems, that monitor power supply units using embedded sensors. Such systems typically transfer large amounts of streaming data; have erratic periods of extreme network activity; are subject to relatively common hardware failures and for comparatively long periods, and; require low jitter and fast response time for safety reasons, e.g., vehicle coordination.

Target Systems

The main motivation for this work was the need to address the requirements of the public transportation solutions at EFACEC, more specifically, the middleware that supports the, non safety-critical, information systems for light-train networks. One such system is deployed at Oporto's light-train network and is composed of 5 lines, 70 stations and approximately 200 sensors. Each station is managed by a computational node, designated as *peer*, that is responsible for managing all the local audio, video, display panels, and low-level sensors such as track sensors for detecting inbound and outbound trains.

The system supports three types of traffic: *normal* - for regular operations over the system, such as playing an audio message in a station through an audio codec; *critical* - medium priority traffic comprised of urgent events, such as an equipment malfunction notification; *alarms* - high priority traffic that notifies critical events, such as low-level sensor events. Independently of the traffic type, the SLAs must be met by EFACEC even under faults, namely: a) any operation must be completed within 2 seconds (e.g., RPC

[*](c) 2012 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the national government of Portugal. As such, the government of Portugal retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

[1]EFACEC, the largest Portuguese Group in the field of electricity, with a strong presence in systems engineering namely in public transportation and energy systems, employs around 3000 people and has a turnover of almost 1000 million euro; it is established in more than 50 countries and exports almost half of its production (c.f. **http://www.efacec.com**).

invocation and event processing), and; b) the system must be able to sustain 200 events per second.

1.1 Challenges and Opportunities

The current architecture at EFACEC uses a rigid centralized, client-server architecture based on a proprietary CORBA implementation that only works for small deployments, i.e., deployments with tens of nodes. For example, the Oporto's light-train system follows a tree-like deployment, with 3 levels of depth, where the root node is composed of the main server and the intermediary level nodes (2nd level) are composed of regional servers with the purpose of aggregating, without redundancy, a variable set of stations (3rd level). As such, in the presence of faults and in overloading conditions, the current solution barely meets the SLAs and does not support fault-tolerance.

At EFACEC we started working on a middleware that could solve the limitations of the current solution. From the point of view of distributed architectures, the current deployments would be best matched with P_2P infra-structures that are resilient and allow resources (e.g., a sensor connected through a serial link to a peer) to be seamlessly mapped to the logical topology. In addition to provide the infra-structure, the overlay should also provide support for soft real-time (RT) tasks and fault-tolerant (FT) services.

RT is required to ensure that SLAs are meet accordingly to the systems requirements, e.g., an alarm must reach an operator within 2 seconds. A failure to meet a RT deadline is handled at the application level, e.g., a video frame received with a latency greater than 10 seconds is considered a failure, while a P_2P reorganization operation fails if it takes more than 1 second to complete.

FT is required to ensure system dependability (and availability) in the presence of faults, e.g., the alarm must reach the operator despite the occurrence of failures.

Moreover, the next generation light-train solutions require deployments across cities and regions that can be overwhelmingly large. Supporting larger deployments in an efficient way requires some form of scalable hierarchical abstraction that takes advantage of the underlying P_2P organization while providing more efficient ways for peers to cooperate and to discover and manage resources. We call this abstraction a *cell*, that is composed of several peers that cooperate to maintain a portion of the mesh, and it is akin, e.g., to Gnutella's ultrapeers [8].

The main challenge in the development of such of a middleware system is that of seamlessly and efficiently integrating real-time and fault-tolerance support. Considering current state-of-the-art research we see many opportunities to address this issue. One is the use of Common Of The Shelf (COTS) operating systems (Linux) that allows for a faster implementation time, thus smaller development cost, while offering the necessary infrastructure to build a new middleware system.

RT support can be achieved by enhancing TAO's [23] and MEAD's[16] threading strategies with resource reservation (e.g., using Linux Control Groups [13] - a low-level resource reservation mechanism implemented by the kernel) and by avoiding traffic multiplexing through the use of different access points to handle different traffic priorities. Currently, our resource reservation infrastructure only supports CPU, but we aim to support memory, network and I/O in the future.

Moreover, in order to provide reliability and availability to our target system, we need to provide FT mechanisms using space redundancy [26], which introduces the need for the presence of multiple copies of the same resource (replicas), and these, in turn, ultimately lead to a greater resource consumption. Thus, FT support can have an important impact on RT tasks. When an operation is performed any state change that it causes must be propagated among the replicas using a replication algorithm that introduces an additional source of latency. Furthermore, the recovery time, that consists in the time that the system needs to recover from a fault, is an additional source of latency to real-time operations.

P_2P networks can be used to provide a scalable and resilient infra-structure that mirrors the physical deployments of our target systems. Furthermore, different P_2P topologies offer different trade-offs between self-healing, resource consumption and latency in end-to-end operations.

Ultimately, by directly implementing FT on the P_2P infrastructure we manage to lower resource usage and latency to allow the integration of RT. We use proven replication algorithms [26, 3] that offer well-known trade-offs regarding consistency, resource consumption and latency, so that we can focus on the actual problem of integrating real-time, fault-tolerance within a P_2P infrastructure.

To the best of our knowledge, Stheno is the first P_2P middleware system, for general purpose computing, that is able to provide scalability, while simultaneously supporting both RT and FT. The contributions of this work include: support for transparent FT directly at the P_2P overlay taking advantage of its resiliency and topology awareness; extension of TAO's and MEAD's support for RT through the enhancement of their threading strategies with kernel based resource reservation (only CPU for now), and; a fully configurable P_2P layer (e.g., mesh topology and algorithms and FT replication algorithms) that provides a good match to the topology of the physical deployments.

Assumptions

The distributed model used in this paper is based on a partial asynchronous computing model, as defined in [3], extended with fault-detectors.

The services and P_2P layer only support crash failures. We consider a crash failure [26] to be characterized as a complete shutdown of a computing instance in the event of a failure, ceasing to interact any further with the remaining entities of the distributed system.

The timing faults are handled differently by services and the P_2P layer. In our service implementations a timing fault is logged (for analysis) with no other action being performed, whereas, in the P_2P layer we consider a timing fault as a crash failure, i.e., if the remote creation of a service exceeds its deadline, the peer is considered crashed. This method is also called as *process controlled crash* [7]. In this work, we adopted a more relaxed version. A peer wrongly suspected of being crashed does not get killed or commits suicide. Instead it gets *shunned*, that is, a peer is expelled from the overlay, and is forced to rejoin it, more precisely, it must rebind using the membership service of the P_2P layer.

The fault model used was motivated by the first author's experience on several field deployments of ligth-train transportation systems, such as the Oporto, Dublin and Tenerife light-train solutions. Due to the use of highly redundant hardware solutions, e.g., power supplies and 10-Gbit network ring links, network failures tend to be short. The most common cause for downtime is related with software bugs, that mostly results in a crashing computing node. While simultaneous failures can happen, they are considered rare events.

We also assume that the resource-reservation mechanisms are always available.

1.2 Related Work

We started by searching for an available off-the-shelf solution that could meet the SLAs from our target systems, or in its absence, identify a current solution that could be extended, and in this way, avoid the creation of a new middleware solution from the ground up. Figure 1 shows the research space that we have covered.

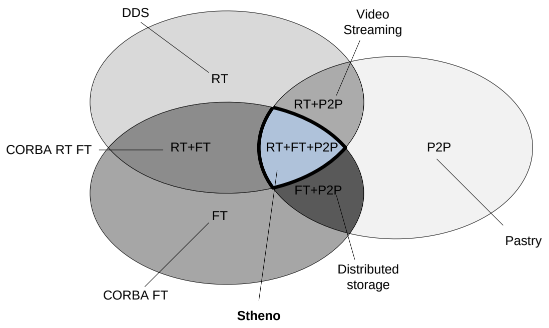


Figure 1: Middleware system classes.

We focused our search on the intersecting domains, namely, RT+FT, RT+P_2P and FT+P_2P, as the systems contained in these domains seemed the most suitable to be extended to support RT+FT+P_2P, as required by our target system.

Within the RT domain, a substantial body of work focused on the integration of real-time within CORBA [23]. More recently, QoS-enabled publisher-subscriber middleware systems based on the JAIN SLEE specification [11] and in the Data Distribution Service (DDS) specification [18], appeared as a way to overcome the current lack of support for real-time applications in SOA-based middleware systems. In the FT domain, CORBA-based middleware systems were a fertile ground to test fault-tolerance techniques, that resulted in the creation of the CORBA-FT specification [19]. Nowadays, some of this focus was redirected to SOA-based platforms, such as JBoss [14].

The support for RT+FT in general purpose distributed platforms remains mostly restricted to CORBA, such as MEAD [16] and TAO [23]. RT support for Java was introduced with the Real-Time Specification for Java (RTSJ) [9], but it was aimed solely to the Java 2 Standard Edition (J2SE). Thus, currently, there is no Java application server (J2EE) able to support RT.

Alternatively, P_2P systems evolved and started to focus on reliability, i.e., FT+P_2P, namely through the introduction of distributed hash tables (DHTs). While some efforts have been made in the past to provide an uniform Application Programming Interface (API) [5] for the various DHTs, e.g., Chord [25], they usually have a tight integration with the target application, not providing a general purpose solution to support a wider range of applications. More recently, these type of systems have been focusing in publish-subscribe systems [20] and storage-based solutions [6].

While most of the focus on P_2P systems has been on the support of FT, there is a growing interest in to the use of P_2P for RT applications, i.e, RT+P_2P, namely in video streaming [10]. Moreover, P_2P systems have been focused on providing Quality-of-Service (QoS) on latency-sensitive applications, such as Video on Demand (VoD) systems [12]

Nevertheless, we were unable to find a suitable middleware system that could be extended in order to simultaneously support RT, FT and P_2P. Systems from the RT+FT domain are not adaptable to the P_2P paradigm, whereas P_2P systems do not offer a general purpose framework that allows to build upon them, forcing us to build a new middleware solution.

In our first effort to support RT+FT+P_2P, DAEM [15], we used some off-the-self components, e.g., JGroups [2] to manage replication groups, but realized that in order to integrate real-time and fault-tolerance within a P_2P infrastructure, we would have to completely control the underlying infrastructure with fine grain management over all the resources available in the system. Thus, the use of COTS software components creates a "black-box" effect that introduces sources of unpredictable behavior and non-determinism that undermines any attempt to support real-time. For that reason, we opted to create a solution from scratch.

The middleware system presented in this work, Stheno, is to the best of our knowledge the first general purpose middleware system that simultaneously supports RT and transparent FT within a P_2P infrastructure.

2. ARCHITECTURE

As we noted above, current state-of-the-art middleware systems [16, 23] address this problem by offering soft real-time computing and fault-tolerance support. Nevertheless, their support for real-time computing is limited, as they do not provide isolation, e.g., a service can hog the CPU and effectively starve the remaining services. The support for FT is normally accomplished through the use of high-level services. Furthermore, enhancing FT with topology awareness requires additional high-level services [1]. However, these high-level services cause a significant amount of overhead, due to cross-layering and long code-paths, limiting the real-time capabilities of these middleware systems.

These systems also used a centralized networking model that is susceptible to single point-of-failure and offers limited scalability. The CORBA naming service is an example of these limitations, where a crash failure can effectively stop an entire system because of the absence of the name resolution mechanism.

Next, we present the architecture of Stheno and show how it addresses the aforementioned problems. The resilient nature and topology awareness of P_2P overlays enables us to overcome the limitations of current approaches by offering a decentralized and reconfigurable fault resistant architecture that avoids bottlenecks, and thus enhances overall performance.

2.1 Stheno's System Architecture

In order to contextualize our approach, we will present our solution applied to one of our target systems, the Oporto's light-train public information system. As shown in Figure 2, the network uses a hierarchical tree-based topology, that is based on the P^3 overlay [17], where each *cell* represents a portion of the mesh space that is maintained (replicated) by a group of *peers*. These peers provide the computational resources needed to maintain the light-train stations and host services within the system. Additionally, there are also *sensors* that connect to the system through peers. These offer an abstraction to several low-level activities, such as traffic track sensors and video camera streams.

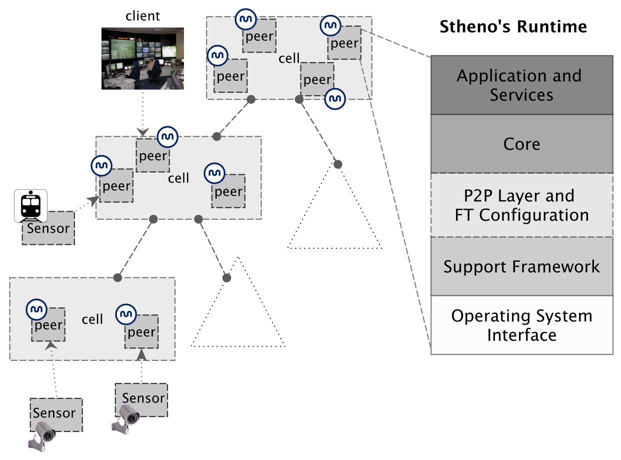


Figure 2: Stheno overview.

The middleware's runtime provides the necessary infrastructure that allows users to launch and manipulate services, while hiding the interaction with the low level peer-to-peer overlay and operating system mechanisms. It is based on a five layer model, as shown in Figure 2.

The bottom layer, *Operating System Interface*, encapsulates the ACE [22] network framework and the Linux operating system. ACE provides abstractions to the operating system calls with the goal of achieving portability. The *Support Framework* is built on top of the bottom layer, and offers a set of high-level abstractions for efficient, modular component design, e.g., threading strategies and the resource reservation infra-structure. The *P_2P Layer and FT Configuration* contains all the peer-to-peer overlay infrastructure components and provides a communication abstraction and FT configuration to the upper layers. The runtime can be loaded with a specific overlay implementation at bootstrap. The middleware is parametric in the choice of the overlay, configurable at bootstrap time. The *Core* layer represents the kernel of the runtime, and is responsible for managing all the resources allocated to the middleware, the peer-to-peer overlay and the middleware services by interacting with a QoS daemon. Note that, while each machine can hold multiple peers from an overlay (instances of the runtime), it runs a single instances of the QoS daemon which is used by all local peers. Finally, the *Application and Services* layer is composed of the applications and services that run on top of the middleware.

2.2 Discussion

Our middleware platform is able to provide QoS computing with support for resource reservation through the implementation of a QoS daemon. This daemon is responsible for the admission and distribution of the available resources among the components of the middleware. It interacts with the low-level resource reservation mechanisms of the operating system to perform the actual reservations. With this support, we provide proper isolation that is able to accommodate soft real-time tasks in order to maximize the probability of meeting the target SLAs. Whilst the use of Earliest Deadline First (EDF) scheduling would provide greater RT guarantees, this goal was not be pursued due to the lack of maturity of the current EDF implementations in Linux (our reference COTS operating system). Because we are limited to use priority based scheduling and resource reservation, we can only partially support our goal of providing end-to-end guarantees, more specifically, we enhance our RT guarantees through the use of RT scheduling policies with over-provisioning to improve the odds that deadlines are met.

While we currently only support CPU reservation, the architecture was designed to be extensible and subsequently support additional sub-systems, such as memory and networking resource reservations.

Notwithstanding, the real-time capabilities are limited by the amount of resources that are needed to provide fault-tolerance. To overcome the current limitations of providing fault-tolerance through the use of expensive high-level services, we propose the integration of the fault-tolerance mechanisms directly in the the overlay layer. This provides two advantages over the previous approaches: a) it allows the implementations of lightweight fault-tolerance mechanism by reducing cross-layering, and; b) the replica placement strategies can be optimized using the knowledge of the overlay's topology. Previous systems relied on manual bootstrap of replicas, such as TAO [23], or required the presence of additional high-level services to perform load balancing across the replica set, as in FLARe [1].

While the work presented in this paper only implements semi-active replication [21], we designed a modular and flexible fault-tolerance infrastructure that is able to accommodate other types of replication policies, such as passive replication [4] and active replication [24].

Given that we wanted to use COTS operating systems and hardware, we used the ACE framework [22] to abstract the underlying operating system infrastructure and therefore minimize the effort required to port the runtime to a new platform.

3. IMPLEMENTATION

We will limit the discussion on Stheno's implementation to our main contributions: real-time support through the extension of TAO and MEAD threading strategies with resource reservation, provided on top of Linux's Control Groups, and the implementation of transparent and topology aware FT mechanisms directly in the P_2P layer.

3.1 Real-Time and Resource Reservation

One of the key aspects of real-time systems is the ability to fulfill a SLA even in the presence of an adverse environment. Adversities such as system overload or bugs can be caused by rogue services, device drivers, or kernel modules.

Our approach to improve support for RT is accomplished by the isolation of the various components present in the runtime through resource reservation. This is achieved by using the Control Groups facility provided by the Linux kernel. Figure 3 shows an overview of Stheno's resource reservation infra-structure.

The *QoS Controller* acts as a proxy between the compo-

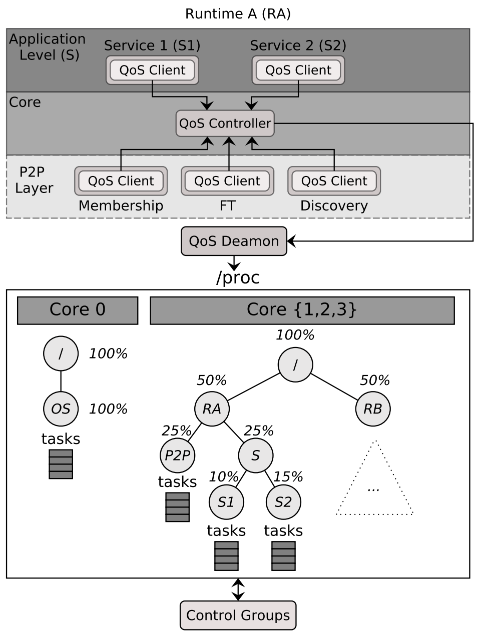


Figure 3: QoS infra-structure.

nents of the runtime and the QoS daemon. Each component has access to resources that are assigned at creation time. A component uses its resources through a *QoS Client*, that was previously assigned to it by the QoS Controller. A resource reservation request is created by a QoS Client and then gets re-routed by the QoS Controller to the QoS daemon. In the current implementation, the allocation of resources is static. A dynamical reassignment of the resources allocated to a component is left for future work.

The goal of the QoS daemon is to provide an admission control and management facility that governs and interacts with the underlying Control Groups infrastructure through the `/proc` pseudo-filesystem. Control Groups supports four main QoS subsystems: CPU, I/O, memory and network. At this time, we only have full support for the CPU subsystem.

All the subsystems supported by Control Groups follow a hierarchical tree approach to the distribution of their resources. Each node of the tree represents a group that contains a set of threads that share the available resources of the group. For example, Figure 3 depicts a possible scenario where a peer has statically partitioned the available CPU cores (in this case a quad-core) between 2 independent partitions. The first partition, composed of core 0, is dedicated to the Linux operating system, while the second partition, composed of cores 1,2 and 3, is dedicated to host Stheno's runtimes. In this case, the available resources were divided equally between two runtimes (runtime B composition is omitted for simplicity), with each one receiving 50% of the CPU time (of cores 1,2 and 3). Looking closely to runtime A, we can see that its resources were also divided equally between the P_2P layer and its services. While we could have further divided the resources allocated to the P_2P layer among its services (FT, discovery and membership), we choose to aggregate them into the leaf designated as "P2P". This was done in order to maximize the available resources to the FT service, i.e., although some contention occurs from this sharing of resources, each individual service has access to a large resource pool. On the other hand, the services allocated to runtime A were isolated from each other, in order to guarantee that they do not interfere.

3.2 P_2P and Fault-Tolerance

As previous stated, we use a P_2P overlay based on the P^3 framework [17] because it best suits our target system, but other topologies, such as Chord [25] or Gnutella [8], can be used as appropriate for the target system.

Within a P^3 cell, peers collaborate to maintain a portion of the overlay. Cells provide resilience to the overlay and improve the efficiency of resource discovery. In each cell there is one coordinator peer. Every other peer in the cell is connected to the coordinator, allowing for efficient group communication. In the case of failure of a coordinator, one of the other peers in the cell takes its place, following an overlay specific algorithm.

The communication between cells is accomplished through point-to-point connections (TCP/IP sockets) between the coordinators. Sensors are not required to help with mesh management. They are usually low resource peers that use the overlay capabilities, for instance, to advertise the availability of a data stream or events.

A P_2P overlay implementation must provide three basic services: the membership, discovery and, fault-tolerance services.

The Membership Service. This service is responsible for the building and maintenance of the mesh by allowing peers to join and leave the overlay. When a peer wants to join the overlay, it first requests the root cell (that acts as a portal to the overlay) to obtain a suitable binding cell. If there is no available cell to accept the incoming peer, then a new cell must be created and the root cell replies with a message containing the coordinator of the parent cell (of the newly created cell) and the new cell identifier. Otherwise, if there is a cell willing to accept the new peer, i.e., a cell that is not completely full, then it joins this active cell. In this case, the reply message contains the cell's coordinator information and also the cell identifier. In both cases, the joining peer connects to the proper coordinator (i.e. the parent coordinator if this is a new cell, or the cell's coordinator if this is an active cell) and sends a join message. This message is propagated through the overlay until it reaches the root cell. It is the responsibility of the root cell to validate the join request and to reply accordingly. The reply is propagated through the overlay downwards to the joining peer. After this, the peer is part of the overlay.

The Discovery Service. The P^3 discovery service uses the hierarchical topology of the mesh to efficiently resolve query requests. When a query is received, the overlay first tries to resolve it locally, and only if this is not possible, it propagates the request to the cell's coordinator. If the coordinator is also unable to reply to the request, the request is propagated once more to its parent coordinator and the process is repeated recursively until a coordinator is able to reply. If this process reaches a point where there is no parent coordinator available, i.e. the root node, the process fails and a failure reply is sent downwards to the originating peer.

The Fault-Tolerance Service. The fault-tolerance service is built on top of the notion of *replication groups*. A replication group can be defined as a set of cooperating peers that have the common goal of providing reliability to a high-level service. Previous work [16, 23], implemented FT support through a set of high-level services that used the underlying primitives of the middleware. Our approach makes a fundamental shift to this principle, by embedding this lightweight FT support at the overlay layer.

Each replication group implements its own replication algorithm. This allows different services to use different replication algorithms, e.g., passive [4], semi-active [21] and active [24], inside the FT service.

The integration of FT in the overlay reduces the overhead of cross-layering that is associated with the use of high-level services. Furthermore, this approach also enables the runtime to make decisions on the placement of replicas that are aware of the overlay topology. This allows a better leverage between resiliency and resource usage. For example, placing replicas in different geographic locations leads to a better resiliency, but can be limited by the availability of bandwidth over WAN links.

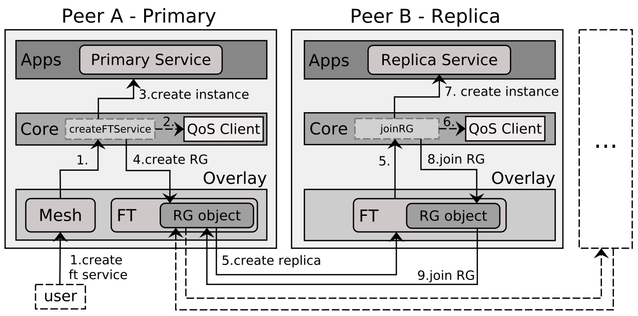


Figure 4: Replication group overview.

Figure 4 shows an overview of the FT service, more specifically, of the bootstrap process of a replicated service. It starts with a peer, in this case referred to as *user*, requesting the creation of a replicated service to peer A, using the membership service. At this point, peer A receives the request (1), and forwards it to its core. Here in a process referred as *createFTService*, it verifies if it is able to host the service. If enough resources are available for hosting the service, that will act as the primary service instance, then the core makes the necessary resource reservations for supporting both the service instance and the replication group (2 and 3). The core delegates to the FT service the responsibility of creating the actual replication group object that will support the replication infrastructure for the service. The type of replication group and the number of replicas depends on the FT parameters embedded in the user's request. Peer A searches for suitable deployment sites in order to match the required number of replicas, using the overlay's discovery service (step omitted). After finding these sites, the primary starts creating the replicas sequentially, using the FT service. The creation of one of these replicas is illustrated in peer B (steps 5 to 9). Upon receiving this request (5), peer B redirects it to its core, where is handled by the process *joinRG*. Peer B does the same procedure as peer A, it checks if the necessary resources are available and makes the resource reservations (6 and 7). Peer B then requests the FT service to join the replication group created by peer A (8). This results in the creation of the local replication group object that makes the final reply to peer A, acknowledging its membership to the replication group (9). This process is repeated in all the peers selected by the discovery service to host replicas.

4. EVALUATION

The physical infra-structure used to evaluate the middleware prototype consists of a cluster of 20 quad-core nodes, each equipped with AMD Phenom II X4 920@2.8Ghz CPUs and 4GB of memory, totaling 80 cores and 80GB of memory. Each node was installed with Ubuntu 10.10 and kernel 2.6.39-git12. The physical network infrastructure was a 100 Mbit/s Ethernet with a star topology.

At bootstrap, the middleware starts by building a peer-to-peer overlay with a user specified number of peers and sensors. The peers are grouped in cells that are created according to the rules of the underlying P^3 overlay. Overlay properties control the tree span and the maximum number of peers per cell at any given depth. For the results gathered, we used a binary tree with 3 levels of depth, with 4 peers in the cell of the first level, 3 peers for the 2 cells in the second level and 2 peers for the 4 cells in the third level.

RPC Benchmark

The RPC service executes a procedure in a foreign address space. This is a standard service in any middleware system. A primary server receives a call from a client, executes it, and updates the state in all service replicas. When all replicas acknowledge the update, the primary server then replies to the client. In the absence of fault-tolerance mechanisms, the primary server executes the procedure and immediately replies to the client.

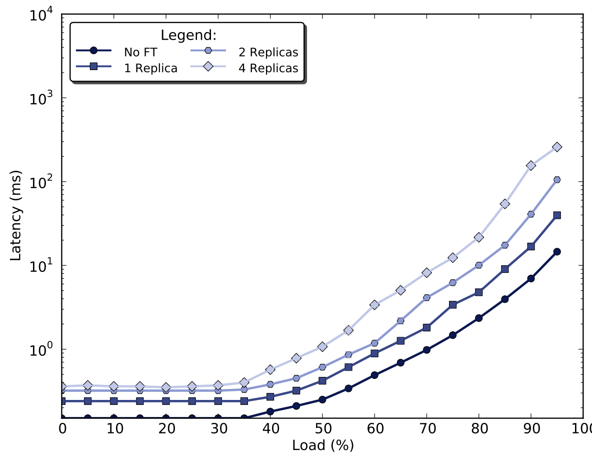
To evaluate the RPC service we used the maximum available priority of 48 (for the services). This was done to prevent starvation of the kernel threads that handle both the low-level Control Groups infrastructure and the interrupt handling. Thus priorities above 48 are only used by the Linux operating system.

The remote procedure simply increments a counter and returns the value. We performed 1000 RPC calls each run, with an invocation rate of 250 per second.

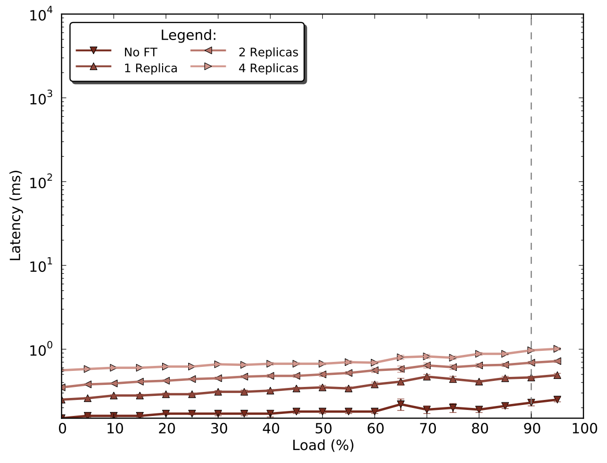
Load Generator

Complex distributed systems can be affected by the presence of rogue services that can become a source of latency and jitter. We evaluate the impact of the presence of such entities by introducing in each peer a *load generator* service. The later spawns as many threads as the logical core count of the CPU. Unless explicitly mentioned, the threads are allocated to the `SCHED_FIFO` schedule class, with priority 48. This scheduling policy represents the worst case scenario of unwanted computation. Given a desired load percentage p (in terms of the total available CPU time), each thread continuously generates random time intervals (up to a configurable maximum of 5ms). For each value it calculates the percentage of time that it must generate load (this is achieved by a set of mathematical operations with the sole purpose of avoiding any optimizations by the compiler) so that it equals p. For example, if the desired load is 75% and the value generated is 4ms, then the load generator must compute for 3ms and sleep for the remainder of that time lapse.

The experiments were tested with increasing load values (5% step), up to a maximum of 95%. For each of these con-



(a) Without resource reservation.



(b) With resource reservation.

Figure 5: End-to-End latency.

figurations we ran the benchmark 16 times, and computed the average and the 95% confidence intervals (represented as error bars). A vertical dashed line corresponding to a load of 90% is used as a reference for the case where resource reservation is enabled.

4.1 End-to-End Latency Results

In this experiment no faults were injected in the system. The invocation latencies without resource reservation depicted in Figure 5(a) show that up to loads of 35% the FT mechanisms introduce low overhead and low jitter. The maximum invocation latency registered, while using 4 replicas, was about 300ms per invocation.

On the other hand, The results from the runs using resource reservation can be seen in Figure 5(b). The fact that the RPC service is now isolated, at least in terms of CPU, from the remainder of the system, contributes to its almost constant latencies and stability (low jitter) with increasing peer loads. The invocation latency also shows the natural increase with the number of replicas. The maximum invocation latency with resource reservation and using 4 replicas was of 1ms per invocation.

4.2 Fail-over Latency Results

The results for the fail-over latency without resource reservation can be seen in Figure 6(a). A single fault was injected in the service at half-time through the execution. The latency values are measured by the client, from the time it first detects the fault until it is able to rebind to the service. In general, the rebind latency presents a stepper increase when compared to invocation latency, although the differences with varying number of replicas are masked by jitter. The rebind process involves several steps: failure detection; election of a new primary server; discovery of new primary server, and; transfer of lost data. In each step, the increasing load introduces a new source of latency and jitter that accumulates to the overall rebind time. In this implementation the client must use the discovery service of the mesh to find the new primary server. This step could be optimized, for example, by keeping track of the replicas in the client. Despite this, the rebind latency remains fairly constant up to loads of 40% to 45%. The maximum fail-over latency without resource reservation was of 3s.

The results for the fail-over latency with resource reservation can be seen in Figure 6(b). With the introduction of resource reservation, the fail-over latencies remain fairly low even with the increase in load. The maximum fail-over latency with resource reservation was below 20ms.

Stheno provides a low end-to-end response time that combined with a low fail-over latency enables it to meet and exceed the target system requirements (2s end-to-end response time), even under the presence of a fault.

5. CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, Stheno is the first system that: a) supports traffic types with different soft-RT requirements; b) supports different FT configurations; c) supports configurability at multiple levels: P_2P, RT and FT, and; d) continues to meet RT requirements even under faults.

Our empirical evaluation shows that Stheno meets and exceeds target system requirements for end-to-end latency and fail-over latency, and thus validates our approach of implementing fault-tolerance mechanisms directly over the peer-to-peer overlay infrastructure as a way to minimize overhead and simultaneously support RT tasks.

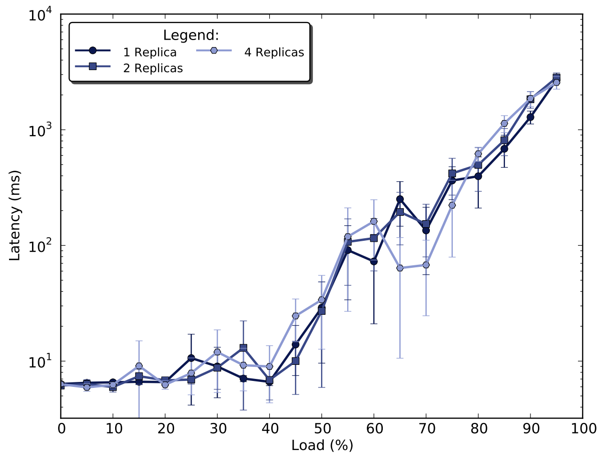
For future work, we would like to evolve from a priority approach to a deadline approach, using the upcoming Linux's EDF scheduler, in order to improve the guarantees on the fulfillment of SLAs.

Acknowledgements. R. Martins was supported by EFACEC Engenharia e Sistemas, S.A. and by FCT PhD grant SFRH/BDE/15644/2006. This work was in part funded by the ERDF/COMPETE Programme and by FCT within project FCOMP-01-0124-FEDER-022701 and project MACAW (PTDC/EIA-EIA/115730/2009).

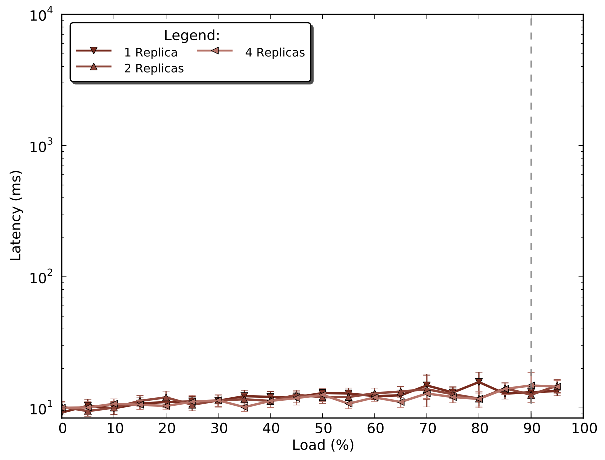
6. REFERENCES

[1] J. Balasubramanian. FLARe: a Fault-tolerant Lightweight Adaptive Real-time Middleware for Distributed Real-time and Embedded Systems. In *Proceedings of MDS'07*, pages 17:1–17:6, New York, NY, USA, November 2007. ACM.

[2] B. Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java. Technical report, Cornell University, Sept. 1998.



(a) Without resource reservation.



(b) With resource reservation.

Figure 6: Fail-over latency.

[3] K. Birman. *Guide to Reliable Distributed Systems.* Texts in Computer Science. Springer, 2012.

[4] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. *The Primary-Backup Approach.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[5] F. Dabek et al. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proceedings of IPTPS*, Berkeley, CA, February 2003.

[6] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of SOSP'07*, pages 205–220, Oct. 2007.

[7] X. Defago. *Agreement-Related Problems: from Semi-Passive Replication to Totally Ordered Broadcast.* PhD thesis, École Polytechnique Fédérale de Lausanne, Aug. 2000.

[8] J. Frankel and T. Pepper. Gnutella Specification. `http://www.stanford.edu/class/cs244b/gnutella_protocol_0.4.pdf`. [Online; accessed 17-10-2011].

[9] J. Gosling et al. *The Real-Time Specification for Java.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[10] Y. Huang et al. Challenges, Design and Analysis of a Large-Scale P2P-VOD System. In *Proceedings of SIGCOMM'08*, pages 375–388, New York, NY, USA, Aug. 2008. ACM.

[11] JCP. JAIN SLEE v1.1 Specification. JCP Document: `http://download.oracle.com/otndocs/jcp/jain_slee-1_1-final-oth-JSpec/`, Jul 2008. [Online; accessed 17-10-2011].

[12] Z. Li et al. QRON: QoS-aware Routing in Overlay Networks. *IEEE Journal on Selected Areas in Communications*, 22(1):29–40, Jan. 2004.

[13] Linux Kernel. Real-Time Group Scheduling. `http://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt`, 2009. [Online; accessed 17-10-2011].

[14] Marc Fleury and others. The JBoss Extensible Server. In *Proceedings of Middleware'03*, pages 344–373, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[15] R. Martins, P. Narasimhan, L. Lopes, and F. Silva. Lightweight Fault-Tolerance for Peer-to-Peer Middleware. In *First International Workshop on Issues in Computing over Emerging Mobile Networks (C-EMNs), Proceedings of SRDS'10*, pages 313–317, November 2010.

[16] P. Narasimhan et al. MEAD: Support for Real-Time Fault-Tolerant CORBA: Research Articles. *Concurrency and Computation: Practice & Experience*, 17(12):1527–1545, October 2005.

[17] L. Oliveira, L. Lopes, and F. Silva. P3: Parallel Peer to Peer An Internet Parallel Programming Environment. In *Web Eng. and P2P Computing*, volume 2376 of *LNCS*, pages 274–288. Springer, 2002.

[18] OMG. OpenDDS. `http://www.opendds.org/`. [Online; accessed 17-10-2011].

[19] OMG. Fault Tolerant CORBA Specification. OMG Technical Committee Document: `http://www.omg.org/spec/FT/1.0/PDF/`, May 2010. [Online; accessed 17-10-2011].

[20] P. Pietzuch et al. Hermes: A Distributed Event-Based Middleware Architecture. In *ICDCS Workshops*, pages 611–618. IEEE Computer Society, Jul. 2002.

[21] D. Powell et al. The Delta-4 Approach to Dependability in Open Distributed Computing Systems. In *Proceedings of FTCS'88*, pages 246–251, Tokyo, Japan, 1988. IEEE Computer Society Press.

[22] D. Schmidt. An Architectural Overview of the ACE Framework. *;login: the USENIX Association newsletter*, 24(1), Jan. 1999.

[23] D. Schmidt et al. The Design of the TAO Real-Time Object Request Broker. *Computer Communications*, 21(4):294–324, 1998.

[24] F. Schneider. *Replication Management using the State-machine Approach.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[25] I. Stoica et al. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM*, volume 31, 4 of *Computer Communication Review*, pages 149–160. ACM Press, Aug. 2001.

[26] P. Veríssimo et al. *Distributed Systems for System Architects.* Kluwer Academic Publishers, Norwell, MA, USA, 2001.