# Lightweight Fault-Tolerance for Peer-to-Peer Middleware

**(extended version of C-EMNs'10 paper)**

Rolando Martins
EFACEC/CRACS & INESC-Porto LA,
Faculdade de Ciências, Universidade do Porto, Portugal
e-mail: rolando.martins@efacec.com

Priya Narasimhan
Carnegie Mellon University
e-mail: priya@cs.cmu.edu

Luís Lopes, Fernando Silva
CRACS & INESC-Porto LA,
Faculdade de Ciências, Universidade do Porto, Portugal
e-mail: {lblopes,fds}@dcc.fc.up.pt

Technical Report Series: DCC-2011-01

---

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

---

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL

Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

# Lightweight Fault-Tolerance for Peer-to-Peer Middleware (extended version of C-EMNs'10 paper)

Rolando Martins*, Priya Narasimhan†, Luís Lopes‡, Fernando Silva‡

*EFACEC/CRACS & INESC-Porto LA, Faculdade de Ciências, Universidade do Porto, Portugal
e-mail: rolando.martins@efacec.com
†Carnegie Mellon University
e-mail: priya@cs.cmu.edu
‡CRACS & INESC-Porto LA, Faculdade de Ciências, Universidade do Porto, Portugal
e-mail: {lblopes,fds}@dcc.fc.up.pt

## Abstract

*We address the problem of providing transparent, lightweight, fault-tolerance mechanisms for generic peer-to-peer middleware systems. The main idea is to use the peer-to-peer overlay to provide for fault-tolerance rather than support it higher up in the middleware architecture, e.g. in the form of services. To evaluate our approach we have implemented a fault-tolerant middleware prototype that uses a hierarchical peer-to-peer overlay in which the leaf peers connect to sensors that provide data streams. Clients connect to the root of the overlay and request streams that are routed upwards through intermediate peers in the overlay up to the client. We report encouraging preliminary results for latency, jitter and resource consumption for both the non-faulty and faulty cases.*

## I. Introduction And Motivation

The development and management of large-scale information systems for application domains that require real-time and fault-tolerant computing is pushing the limits of the current state-of-the-art in middleware frameworks. Requirements for faster development cycles, software reuse, greater scalability, dependability and maintenance motivate the research and use of decentralized middleware-based architectures to serve as mediators between applications and the underlying operating systems, network protocol stacks, and hardware.

Fault-tolerance support makes such systems capable of detecting certain categories of failures (e.g. hardware failure, network connectivity) and recover from these without, or with minimal, disruption of services. Real-time support, on the other hand, allows for the implementation of time critical operations that must be completed with a high level of priority or within a rigid time frame. To provide for such features a system must be resource aware and capable of directly controlling resource distribution at a very fundamental level (e.g. network bandwidth, CPU reservation). Middleware systems that combine both of these features pose quite formidable implementation challenges given the difficulty in managing system resources for real-time tasks in the presence of resource hungry fault-tolerance services.

Research in the integration of fault-tolerance and real-time has focused mostly on CORBA and its siblings, for which specifications have been proposed for Fault-Tolerant [13, 14] and Real-Time [9, 15, 17] support. Their integration, however, is difficult. The idea behind FT-CORBA is to implement the fault-tolerance mechanisms as a service or a set of services within CORBA itself. The advantage of this approach is that it provides transparent (i.e. network independent) fault-tolerance mechanisms. The price however is high, as it introduces overhead due to cross-layer service protocols, long code paths and resource consumption. The additional overhead makes the integration of real-time support difficult or impossible. MEAD [12] provides transparent fault-tolerance at a somewhat lower-level using *interceptors* that capture IIOP messages between the applications and the ORB, and redirect them to internal replication and logging-recovery managers that provide fault-tolerance mechanisms. Another approach is followed by TAO [18] that uses request redirection in a strict client-server fashion to implement non-transparent

fault-tolerance mechanisms. In all these systems, the strict adherence to the client-server paradigm severely limits their overall scalability.

In peer-to-peer networks research has focused on architecture, protocols and algorithms that address fault-tolerance, QoS, and some aspects of real-time, for distributed file-sharing systems [5, 6, 10, 11]. Some other work has focused on the specialized case of providing real-time video and/or audio streaming capabilities [8].

In this paper we argue that a lightweight implementation of fault-tolerance mechanisms in a middleware is fundamental for the successful integration of soft real-time support. Our approach is novel in that it explores peer-to-peer networking as a means to implement generic, transparent, lightweight fault-tolerance support. We do this by directly embedding fault-tolerance mechanisms into peer-to-peer overlays, taking advantage of their scalable, decentralized and resilient nature. For example, peer-to-peer networks readily provide the functionality required to maintain and locate redundant copies of resources. Given their dynamic and adaptive nature, they are promising infra-structures for developing lightweight fault-tolerant and soft real-time middleware.

## II. Problem Statement

Our goal is to validate the thesis that using low-level networking from peer-to-peer overlays in a middleware architecture enables the implementation of transparent, generic and lightweight fault-tolerance mechanisms. We present a prototype for such a system based on a hierarchical peer-to-peer overlay, $P^3$ [16], in the lines of Gnutella [7] and P-Grid [1]. Using a worst case scenario overlay configuration, we estimate latency, jitter and resource consumption for the non-faulty case to establish a baseline for the overhead associated with the fault-tolerance mechanism. We also evaluate latency and jitter in the presence of incremental random peer failures (crashes).

The remainder of the paper is structured as follows. Section III describes the overlay architecture used in the middleware prototype. Section IV describes the experimental setup, the metrics used for evaluation, and the results we have obtained. Section V ends the paper with the conclusions and a brief description of current work.

## III. The Overlay Architecture

This section describes the networking layer of the prototype middleware used to validate our fault-tolerance implementation. The prototype has been implemented in Java.

### A. Architecture

The networking layer of the middleware is a peer-to-peer overlay based on the $P^3$ framework [16] (Figure 1).

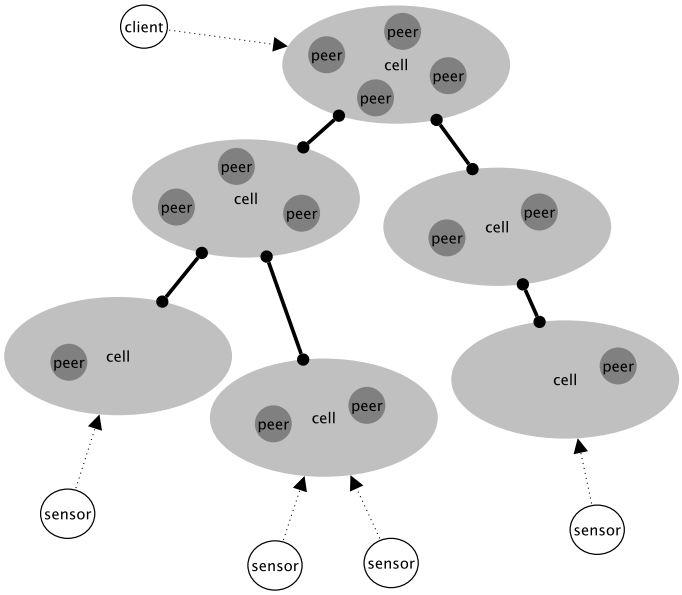


**Fig. 1. A $P^3$ peer-to-peer overlay.**

The peers are responsible for maintaining the organization of the overlay and for providing access points to the overlay for external peers. Peers in the overlay may connect to external sensors dedicated to data collection and its dissemination. Client peers act as data sinks in the network. Requests for data with given QoS parameters (e.g. maximum latency) are issued from client peers and the overlay routes the relevant data packets from the sensor peers up to the client peers, where it is eventually processed. Each node in a $P^3$ network corresponds to a cell, a set of peers that collaborate to maintain a portion of the overlay. Cells are logical constructions that provide overlay resilience and are central in our implementation of fault-tolerance mechanisms. Thus communication between distinct cells is accomplished through point-to-point connections (TCP/IP sockets) between peers in those cells. Communication between peers within a cell uses JGroups [2], a reliable group communication framework, to ensure strong replica consistency for fault-tolerance.

Each peer provides two basic services. The *overlay service* handles the traffic generated for the management of the peer-to-peer overlay. In a sense it is the most fundamental service since it provides the infra-structure for the other service, the *streaming service*. The later is responsible for the relay of data in the overlay and includes built-in fault-tolerance mechanisms. Overlay and

data traffic each use a dedicated communication channel. The overlay service has two sub-services: membership and discovery. Resource discovery in $P^3$ is simplified through the use of a naming scheme for cells and peers that maps in a simple way to their position in the overlay.

## B. Run-Time Semantics

We now describe the basic run-time operations of the peer-to-peer overlay: the construction and management of the overlay, the establishment of client requests, the routing of data streams, and fault-detection and recovery.

*a) Building, membership and discovery:* The membership mechanism allows a peer to join the peer-to-peer overlay (Figure 2). The process starts with a request for a binding cell. This request has to be made to the root cell, that in turn replies with a suitable cell. Upon receiving the reply, the joining peer binds to the cell. After a successful bind, the peer must now request a parent peer in the cell immediately above the current in the overlay hierarchy. If it manages to obtain a parent peer and to successfully bind to it, then the peer sends a join message with the newly found parent. 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.

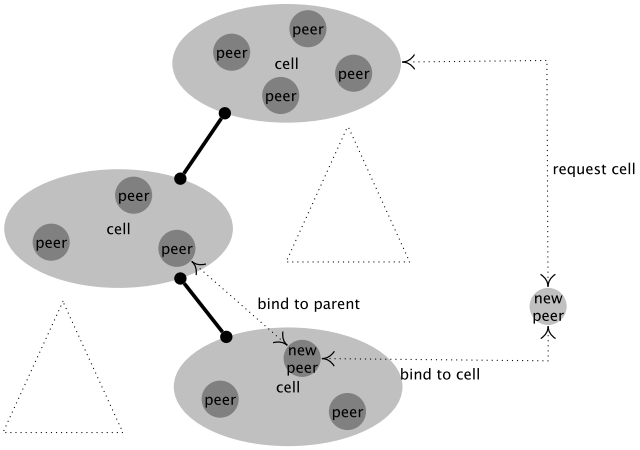


**Fig. 2. Binding to the overlay.**

A peer may also part with the overlay if its parent peer fails or leaves the overlay for some reason. If this situation arises, the peer uses the discovery service to find a new suitable parent in the same cell of the original parent. If this operation succeeds the peer is again part of the overlay, otherwise it must restart the binding process described above.

If all the peers in a cell fail then the peer-to-peer overlay must re-structure itself, in order to maintain the connectivity of the cell's sub-tree.

*b) Client requests:* Client peers may connect to the overlay, through the root, to request a data stream from one or more of the sensor peers. When such a request is made the discovery service is used to find the appropriate sensor in the overlay. Afterwards, a path is established from the root of the overlay to the sensor (Figure 3). Client requests not only select data streams but also specify QoS parameters associated with the request. These may be real-time parameters such as the maximum allowed latency for packets arriving from a sensor, or fault-tolerance related parameters such as whether or not the packets should be replicated, the type of replication, and the number of replicas.

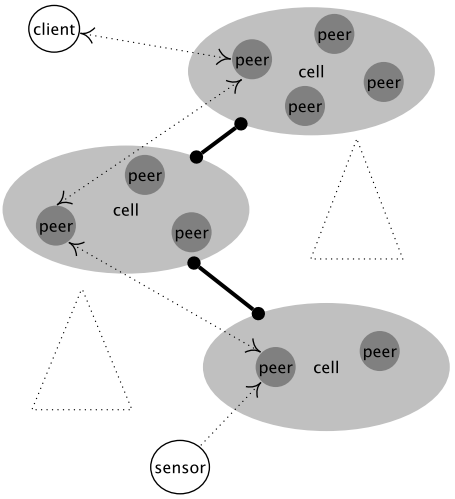


**Fig. 3. Establishing a client-sensor path.**

The need to establish this client-sensor path stems from the fact that we are interested in later integrating real-time support in the middleware. This will have the form of CPU reservation mechanisms for real-time tasks in the peers, and bandwidth reservation protocols along paths in the overlay (involving peers and routers). To do this we must have full control of the hardware resources along any client-sensor path. In this paper, however, we are just trying to evaluate the latency and resource usage due to the built-in fault-tolerance mechanisms in the overlay and whether this precludes the future integration of real-time support.

*c) Routing and replication:* When a path is established between a client and a sensor, each intermediate peer that forms the path in the overlay is used to relay data (Figure 4). In the simplest case, when no replication is required, the peers just forward the packets to the next

peer in the path. If the traffic must be replicated, then each packet generated by a sensor is tagged for replication, and includes a unique key. Every intermediate peer in the path to the client takes the packet and the key and distributes copies to a number of peers in the same cell (the number of replicas is a QoS parameter also). This operation is done using JGroups [2] to ensure strong replica consistency within the cell (Figure 4, left). The use of this tool was a compromise between a less than ideal performance and fast prototyping. Also, the moment when this copy is performed depends on which type of replication is being used (another QoS parameter), that is detailed below. Finally, the peer forwards the packet and the key to the next peer in the path. Each peer that keeps a copy of the packet also maintains a hash table with the corresponding keys.

For this study we have implemented two replication strategies: *active replication* (based on [19]), and passive replication (based on [4]). Other strategies have been proposed like the semi-active replication strategy present in Delta-4/XPA [3] and hybrids between active and passive replication [12], but here we will only consider the aforementioned two. In its original form, the first strategy consists in replicating the state of the service among all non-faulty replicas, with the requests being served in the same order in all of them. Thus, in our middleware, active replication replicates the packets immediately on arrival to the peer. The second strategy originally allocates one primary replica while the all the others act as backups. A client only communicates with the primary replica, that in turn periodically updates its state with the state in the backups. Thus, in our implementation of passive replication, upon arrival the packets are stored in the peer and periodically sent to the backup replicas. The period is a QoS parameter of the system.

When a packet is received by the client, an acknowledgment message with the key is sent in the opposite direction, down the path. Each intermediate peer receives the message and propagates it to the peers in the same cell that hold copies of the message associated with the key. This propagation is regulated by the type of the replication strategy used. If active replication is used, the acknowledgment is sent immediately to the cell and each peer removes its copy of the message. Otherwise, if passive replication is used, the peer that receives the acknowledgment message, searches its buffered messages for a matching key. If the key is found, the associated message is removed (Figure 4, right). If the period of replication is such that the acknowledgement reaches the peer before the primary copy has been replicated in the cell, significant processing and memory savings can be achieved, although message losses may increase as a result of the delayed replication (c.f. Section IV).

After negotiating the replication procedure inside the cell, the peer then forwards the acknowledgment message down to the next peer in the path. Eventually the acknowledgment reaches the peer that first propagated the data packet whose reception is being acknowledged by the client and the send is completed. In fact, after sending a data packet, these *originating* peers wait for the corresponding acknowledgment until a timeout is reached. If the timeout is reached the peer retries to send the data a predefined number of times. The timeout for each subsequent try is increased to allow for latency in the overlay.

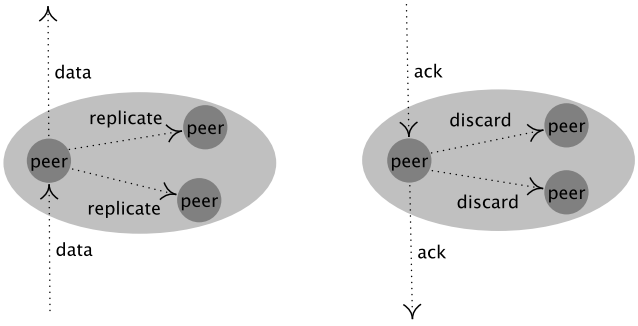


**Fig. 4. The data routing semantics.**

*d) Fault detection and recovery:* While a client request is being serviced, peers in the overlay along the established client-sensor path may fail. In this paper we model only complete peer failures (crashes), from which peers never recover. The prototype however also supports other models, e.g. byzantine failures. When a peer fails, its sibling peers in the cell and its parent peer detect that the connection is down (Figure 5, left). This is implemented by doing periodic pings with timeouts over the TCP/IP sockets connecting the peers. When a fault is detected, a replacement for the faulty peer is elected among the peers that have copies of the messages relayed by it. The process uses the Levenshtein distance between the peer identifiers in the overlay [20]. The peer that computes the minimum distance is elected. This peer recovers the copies of the pending messages of the faulty peer and re-sends them, thus assuming the original peer's role in the client-sensor path (Figure 5, right). It is possible, although very unlikely, that more than one peer is elected in this process. This will just result in duplicate messages being delivered and these will be discarded by the client.

In our prototype the recovery of data can only occur after it enters the overlay. If the peer that directly connects to the sensor in the path fails, some data will be irrecoverable. The sensor will continue to stream data while the overlay establishes an alternative path between the sensor and the client. The amount of data lost depends on how

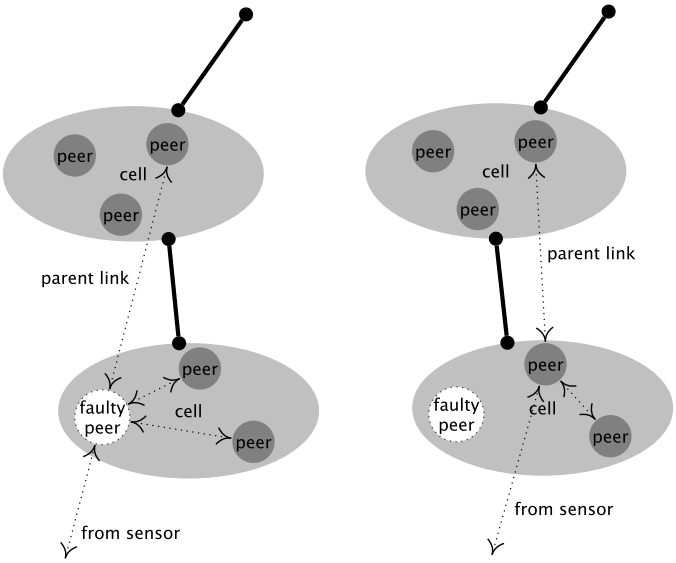


**Fig. 5. Fault detection and recovery.**

fast the overlay detects and recovers from the fault.

## IV. Validation

The physical infra-structure we used to evaluate the prototype consists of a cluster of four identical nodes, equipped with Intel Corei7 920@2.67Ghz CPUs and 6Gb of memory, totaling 16 cores and 24Gb of memory. The physical network infrastructure was based on a 1 Gbit/s Ethernet network with a star topology. The clients, peers and sensors are each executed in a Java Virtual Machine (JVM). The clients and sensors are located in the same machine as this allows the determination of the one-way latency and jitter for the traffic (Figure 6). The prototype 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 $P^3$ framework. Overlay properties control the tree span and the maximum number of peers per cell, at a given depth. In the experiments below, we use a overlay with a depth of 3 (levels 0, 1, and 2), arranged as a binary tree with 2 peers per cell, totaling 14 peers. Each of the 8 peers at level 2 is also connected to a single sensor. Finally, a client connected to the root cell acts as the data sink, giving a full total of 23 JVMs (running on a total of 16 cores). The data generated by the sensors is composed of 3 types of packets: small packets (38 fps, 418 bytes), medium packets (25 fps, 1024 bytes) and large packets (24 fps, 4180 bytes). The sizes and frame rates are based on typical values for audio, events and video. The packets are mixed in the following percentages in the total traffic: 40%, 30% and 30%, respectively.

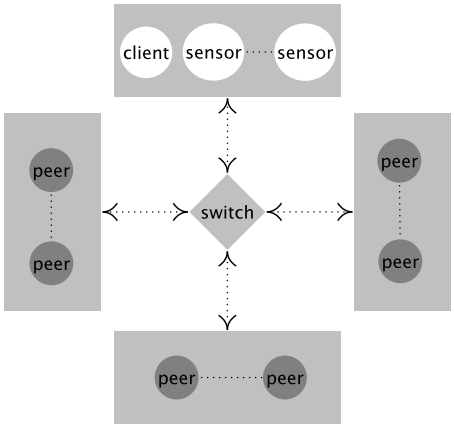


**Fig. 6. The experimental setup.**

*e) Latency, Jitter and Resource Usage – Non-Faulty Case:* We evaluate the overhead induced by the fault-tolerance mechanisms in the transference of data from sensors to clients through the overlay. We do this for a worst-case scenario: a client requests the data from up to 8 sensors connected to the overlay. With this basic scenario we measure the latency, jitter (per packet type) and resource usage (total) in the absence of faults, so that only the overhead of the routing and replication mechanisms is measured. Also, in these runs we replicate each packet in all peers of each cell along the path. This again is a worst-case scenario as typically the number of replicas can be substantially lower. The following cases are studied: 0 streams with fault-tolerance activated and 8 without (0/8), (1/7), (4/4) and the most demanding case (8/0) in which all streams are replicated. The results can be seen in Table I.

| | Packet | Latency(ms) | CPU (s) | Mem (MB) |
|---|---|---|---|---|
| 0/8 | large | 1.70 ± 0.06 | 9.29 ± 5.95 | 35.36 ± 0.71 |
| | medium | 1.94 ± 0.08 | | |
| | small | 1.82 ± 0.07 | | |
| 1/7 | large | 1.94 ± 0.05 | 13.21 ± 8.61 | 39.72 ± 5.0 |
| | medium | 2.18 ± 0.07 | | |
| | small | 2.08 ± 0.07 | | |
| 4/4 | large | 2.58 ± 0.04 | 19.69 ± 11.52 | 61.62 ± 21.03 |
| | medium | 2.76 ± 0.05 | | |
| | small | 2.58 ± 0.07 | | |
| 8/0 | large | 3.38 ± 0.07 | 28.92 ± 11.58 | 91.37 ± 35.03 |
| | medium | 3.58 ± 0.07 | | |
| | small | 3.36 ± 0.05 | | |

**TABLE I. Worst-case scenario latency and resource usage.**

The table shows that an eightfold increase in traffic handled by fault-tolerance mechanisms (case 0/8 vs. 8/0)

results in approximately a twofold increase in packet latency and jitter. In fact, the latency apparently grows logarithmically on the number of streams replicated. The impact in the total CPU and peak memory usage is more important with an approximate threefold increase. The large standard deviation in the memory values results in part from the fact that these are peak memory usage values provided by the JVM. Estimating the total memory usage is difficult due to the lack of control over the JVM's internal activities. Another factor is the asymmetry in memory usage between the peers at the bottom and the peers at the top levels of the tree, the later processing far more incoming traffic and therefore more memory. A significant part of this CPU and memory usage is due both to JGroups and to the large number of replicas used. There is much room for improvement here and it is important that the overhead seems to scale favorably with the number of sensors.

*f) Latency, Jitter – Faulty Case:* Table II shows the aggregated latency (average of the three packet types) and jitter in the presence of peer faults (crashes) for the previous configurations 0/8 and 8/0. The faults are introduced randomly during the run at a predefined level of the overlay. A fault can affect an entire cell, depending on the location of the faulty peer. For example, a failure of two peers can induce a cell failure if the cell is composed of just these two peers. Otherwise, the faulty peers may belong to different cells and each have siblings to take over their traffic management chores. Since the faults are random the values in the table represent an average of both scenarios.

| **Level 0** | | | | |
|---|---|---|---|---|
| **#Faults** | **1** | **2** | | |
| 0/8 | 1.68±0.04 | 1.84±0.08 | | |
| 8/0 | 3.54±0.29 | 6.63±1.26 | | |
| **Level 1** | | | | |
| **#Faults** | **1** | **2** | **3** | **4** |
| 0/8 | 2.0±0.06 | 1.95±0.11 | 1.89±0.14 | 1.88±0.11 |
| 8/0 | 4.05±0.37 | 4.08±0.42 | 4.31±0.33 | 4.73±0.69 |
| **Level 2** | | | | |
| **#Faults** | **1** | **2** | **3** | **4** |
| 0/8 | 1.84±0.05 | 1.82±0.12 | 1.91±0.12 | 1.86±0.09 |
| 8/0 | 3.68±0.20 | 3.75±0.31 | 3.86±0.21 | 3.61±0.24 |
| **#Faults** | **5** | **6** | **7** | **8** |
| 0/8 | 1.83±0.03 | 1.86±0.11 | 1.75±0.06 | 174±0.07 |
| 8/0 | 3.62±0.21 | 3.90±0.19 | 3.74±0.38 | 3.56±0.23 |

**TABLE II. Latency and jitter in the presence of faults.**

The table shows that the overlay copes with many failures and still maintains its responsiveness. The average overhead is fairly constant, approximately a twofold increase relative to the baseline case with fault-tolerance deactivated, across the different levels of the overlay. Here we assume that a packet is lost if it does not arrive at the client until the end of the run. The losses associated with the experiments in table II are rather small, around 1%. More stringent QoS requirements associated with client requests (e.g. maximum packet latency) would likely increase the packet losses. This, however, is a matter for future work on the integration of soft real-time in the middleware system.

*g) Active vs Passive Replication:* The latency times are a good measure of the overhead brought by the different replication strategies. Table III presents the packet latency when injecting faults at different levels of the tree in the presence of active replication, passive replication (with two test-cases, one with 100ms period and the other with 1s period) and no fault-tolerance.

| | **Level 0** | **Level 1** | **Level 2** |
|---|---|---|---|
| No FT | 1.68±0.04 | 2.00±0.06 | 1.84±0.05 |
| Passive-FT (1s) | 3.43±0.32 | 3.86±0.42 | 3.54±0.13 |
| Passive-FT (0.1s) | 3.38±0.23 | 3.87±0.19 | 3.61±0.20 |
| Active-FT | 3.54±0.29 | 4.05±0.37 | 3.68±0.20 |

**TABLE III. Packet latency with one fault per level.**

The latency with active replication is at most 5% greater than with passive replication. This can be explained by the fact that, with active replication, each message that arrives to a peer must be synchronized with the replication group within that cell. The packet may only be forwarded to the parent cell in the mesh after this internal synchronization is performed, and this is the dominant contribution for the overall communication latency.

In the passive replication approach, the newly arrived message is queued for periodic replication within the cell. Despite some contention due to concurrency in the access to the queues, the amount of latency introduced is minimal when compared with active replication.

The results for the resource usage for both active and passive replication strategies, for a scenario similar to that described for Table III, are presented in Table IV.

The resource usage for active replication needs approximately more 20% CPU time, and approximately 15% more of memory than the test scenario with passive replication with 1 second period. This variation is explained by the workload put on the underlying synchronization mechanism, JGroups. If in the case of active replication, we synchronized each message individually, in the passive approach we batch fault-tolerance messages, thus reducing dramatically the amount of calls made to JGroups. In terms of memory usage, the passive replication strategy has a clear advantage over active replication which is due to the fact that a message is not immediately replicated to the cell upon arrival to a peer, but is immediately propagated upwards towards the client. If the acknowledgement of

| Level | Fault at Level 0 | | | | | |
|---|---|---|---|---|---|---|
| | 0 | | 1 | | 2 | |
| | CPU (s) | MEM (MB) | CPU (s) | MEM (MB) | CPU (s) | MEM (MB) |
| No-FT | 36.08±2.42 | 69.09±1.10 | 34.06±0.71 | 141.06±0.91 | 49.51±0.76 | 284.33±0.49 |
| Passive-FT (1s) | 71.09±3.20 | 316.81±2.75 | 77.07±1.68 | 349.77±14.29 | 123.92±3.60 | 522.05±7.26 |
| Passive-FT (0.1s) | 76.43±3.84 | 295.33±5.60 | 94.99±4.62 | 352.33±17.13 | 160.66±3.33 | 509.95±11.22 |
| Active-FT | 92.24±4.52 | 343.57±15.56 | 112.23±2.61 | 388.40±21.18 | 185.24±1.47 | 559.67±15.20 |
| **Level** | **Fault at Level 1** | | | | | |
| | 0 | | 1 | | 2 | |
| | CPU (s) | MEM (MB) | CPU (s) | MEM (MB) | CPU (s) | MEM (MB) |
| No-FT | 46.20±2.84 | 71.63±0.85 | 31.85±3.11 | 140.27±1.25 | 51.34±3.22 | 282.36±4.87 |
| Passive-FT (1s) | 89.43±8.81 | 296.57±8.53 | 74.42±2.69 | 356.01±23.2 | 116.14±3.40 | 505.54±13.12 |
| Passive-FT (0.1s) | 97.16±12.12 | 294.41±3.47 | 89.09±3.44 | 349.83±20.99 | 155.35±2.74 | 519.48±21.01 |
| Active-FT | 114.02±7.31 | 339.98±19.50 | 102.96±3.73 | 362.46±23.01 | 181.06±4.47 | 601.37±23.72 |
| **Level** | **Fault at Level 2** | | | | | |
| | 0 | | 1 | | 2 | |
| | CPU (s) | MEM (MB) | CPU (s) | MEM (MB) | CPU (s) | MEM (MB) |
| No-FT | 42.3±5.34 | 71.32±1.49 | 34.8±1.96 | 140.41±1.30 | 48.99±1.50 | 284.16±1.06 |
| Passive-FT (1s) | 82.01±3.73 | 303.71±5.13 | 79.37±4.86 | 344.63±14.34 | 114.58±5.63 | 508.12±15.50 |
| Passive-FT (0.1s) | 90.26±3.33 | 291.2±2.82 | 97.25±4.63 | 352.96±8.77 | 148.6±2.73 | 504.72±16.29 |
| Active-FT | 107.17±2.17 | 355.44±29.44 | 115.14±5.31 | 389.61±38.39 | 172.87±4.28 | 562.39±21.98 |

**TABLE IV. Resouce usage in the presence of one fault per level.**

reception from the client arrives before a replication period has elapsed, a peer along the path to the sensor may delete its copy of the message before it is replicated further to the other peers in the cell.

Active replication can be seen as a limit case of passive replication, in which the replication period is set to 0s. Indeed, observing the table we see that there is a trend towards increasing CPU usage with the period varying from 1s, 0.1s, and 0s. This is due to the fact that, with decreasing period it gets more and more difficult for the acknowledgment for a message to reach the originating peer before the period elapses and replication through the cell (with JGroups) is triggered. With memory usage we have essentially a tie. In fact, one might expect that the amount of memory used would increase with decreasing period since more replicas of the data would be produced. However, at least for the periods of 1s and 0.1s the difference is not statistically meaningful. On the other hand, memory usage for active replication is clearly above either of the passive replication cases, as expected.

## V. Conclusions and Future Work

In this paper we addressed the problem of implementing transparent, lightweight fault-tolerance mechanisms for middleware systems. We propose that peer-to-peer overlays provide good infra-structures upon which such mechanisms can be implemented, taking advantage of their de-centralized and resilient nature and discovery services.

We present a prototype implementation of a peer-to-peer overlay with built-in fault-tolerance support. We report some preliminary measures of latency, jitter and resource consumption for the non-faulty case to establish a baseline for the overhead associated with the fault-tolerance mechanisms in a worst case scenario. We also evaluate the evolution of latency and jitter in the presence of incremental random peer failures. The results show that even in the worst case scenario the overhead scales favorably with the number of replicated streams and much improvement is possible.

Currently we are implementing the middleware platform in C++ and using built-in protocols for strong-replica consistency (as opposed to using a tool in the lines of JGroups). This implementation will also give us an additional level of control over the hardware (e.g. CPU reservation and scheduling) essential for the future integration of real-time support.

**Acknowledgements** Luís Lopes and Rolando Martins are partially supported by project CALLAS of the Fundação para a Ciência e Tecnologia (contract PTDC/EIA/71462/2006). Rolando Martins is also supported by EFACEC - Sistemas de Electrónica, S.A. and by PhD grant SFRH/BDE/15644/2006 from the Fundação para a Ciência e Tecnologia.

## References

[1] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: a Self-Organizing Structured P2P System. *SIGMOD Rec.*, 32(3):29–33, 2003.

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

[3] P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, D. Seaton, N. Speirs, and P. Veríssimo. The delta-4 extra performance architecture (xpa). In *Digest of Papers of the 20th IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, pages 481–488, jun 1990.

[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, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 202–215. ACM Press, October 2001.

[6] P. Druschel and A. Rowstron. PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of 8th Workshop on Hot Topics on Operating Systems (HotOS VIII)*, pages 75–80, 2001.

[7] J. Frankel and T. Pepper. Gnutella Specification. http://gnet-specs.gnufu.net/.

[8] G. Tan and S. A. Jarvis. Improving the Fault Resilience of Overlay Multicast for Media Streaming. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):721–734, 2007.

[9] A. Gokhale, B. Natarajan, D. Schmidt, and J. Cross. Towards Real-Time Fault-Tolerant CORBA Middleware. *Cluster Computing*, 7(4):331–346, 2004.

[10] J. Kubiatowicz, D. Bindel, Y. Chen, S. E. Czerwinski, P. R. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX), ACM SIGPLAN*, pages 190–201. ACM Press, 2000.

[11] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI'02)*, Operating Systems Review, pages 31–44. ACM Press, December 2002.

[12] P. Narasimhan, T. A. Dumitraş, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava. MEAD: Support for Real-Time Fault-Tolerant CORBA: Research Articles. *Concurrency and Computation: Practice & Experience*, 17(12):1527–1545, 2005.

[13] B. Natarajan, A. Gokhale, S. Yajnik, and D. Schmidt. DOORS: Towards High-Performance Fault Tolerant CORBA. In *International Symposium on Distributed Objects and Applications (ISDOA-00)*, pages 39–48, 2000.

[14] Object Management Group. Fault Tolerant CORBA Specification. OMG Technical Committee Document, June 2002.

[15] Object Management Group. Real-time CORBA Specification. OMG Technical Committee Document, January 2005.

[16] L. Oliveira, L. Lopes, and F. Silva. $P^3$: Parallel Peer to Peer – An Internet Parallel Programming Environment. In *Workshop on Web Engineering & Peer-to-Peer Computing, part of Networking 2002*, volume 2376 of *Lecture Notes in Computer Science*, pages 274–288. Springer-Verlag, 2002.

[17] D. Schmidt and F. Kuhns. An Overview of the Real-Time CORBA Specification. *IEEE Computer*, 33(6):56–63, 2000.

[18] D. Schmidt, D. Levine, and S. Mungee. The Design of the TAO Real-Time Object Request Broker. *Computer Communications*, 21(4):294–324, 1998.

[19] F. B. Schneider. *Replication management using the state-machine approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[20] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Sov. Phys. Dokl.*, 6:707–710, 1966.