On the Impact of Fault-Tolerance Mechanisms in a Peer-to-Peer Middleware

Rolando Martins
EFACEC/CRACS & INESC-Porto LA,
Universidade do Porto, Portugal
Email: rolando.martins@efacec.com

Priya Narasimhan
Carnegie Mellon University
Email: priya@cs.cmu.edu

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

Technical Report Series: DCC-2010-02

On the Impact of Fault-Tolerance Mechanisms in a Peer-to-Peer Middleware

Rolando Martins
EFACEC/CRACS & INESC-Porto LA,
Universidade do Porto, Portugal
Email: rolando.martins@efacec.com

Priya Narasimhan
Carnegie Mellon University
Email: priya@cs.cmu.edu

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

***Abstract*— We address the problem of integrating real-time fault-tolerance mechanisms into peer-to-peer systems, with specific architecture and deployment constraints. For this purpose we implemented a prototype hierarchical peer-to-peer framework in which the leaf peers are sensors that generate different kinds of traffic such as mesh management, events, video and audio. We evaluate the framework by measuring packet loss, response time, jitter and mesh overhead for each type of traffic under peer failure. We report significant gains in QoS for all types of traffic, when using the fault-tolerance mechanisms, with minimal response time and mesh management overhead.**

***Abstract*— We address the problem of integrating real-time fault-tolerance mechanisms into peer-to-peer systems, with specific architecture and deployment constraints. For this purpose we implemented a prototype hierarchical peer-to-peer framework in which the leaf peers are sensors that generate different kinds of traffic such as mesh management, events, video and audio. We evaluate the framework by measuring packet loss, response time, jitter and mesh overhead for each type of traffic under peer failure. We report significant gains in QoS for all types of traffic, when using the fault-tolerance mechanisms, with minimal response time and mesh management overhead.**

I. Introduction And Motivation

Distributed computing systems are becoming larger, more complex, more heterogeneous and more pervasive. In particular, 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 protocols stacks, and hardware.

"Whatever can go wrong will go wrong, and at the worst possible time, in the worst possible way at a blink of an eye. So imagine that in a train wreck!", Adaptation of Murphy's Law. Every system is vulnerable to faults, as they could come from natural disasters, hardware failures or software bugs, just to name a few. The railroads systems are no exception, thus our goal is to ensure that the information flows in the system are delivered in deterministic time and always in a consistent manner. For achieving those goals, a infrastructure must be able to be resilient, in a way that is able to detect faults, recover from them, while maintaining deterministic behavior.

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 problems given the difficulty in integrating, into a single framework, all this level of control. The problem that this paper addresses is the seamless integration of fault-tolerance and real-time, and all the necessary mechanisms needed to accomplished that.

Our approach is novel in that explores the networking layer as a means to implement more lightweight fault-tolerance support, and still accommodate the necessary resource reservation mechanisms to provide real-time support. In this paper we explore this idea by directly embedding fault-tolerance mechanisms into peer-to-peer meshes, taking advantage of their decentralized and resilient nature. For example, peer-to-peer networks readily provide the infra-structure required to maintain and locate redundant copies of resources. Given its dynamic and adaptative nature, we think they are the most suitable candidate for this type of systems.

Research in fault-tolerance has focused mainly in providing mechanisms that are able to couple with faults, normally, from a high level perspective incurring in an excessive overhead from cross-layering. Nevertheless, the principal obstacle for the development of more efficient fault-tolerance mechanisms lies in the architectural adoption of the client-server paradigm, that limits the overall scalability of such systems. The amount of resources needed by these fault-tolerance mechanisms and the non-seamless integration of real-time, significantly limits the introduction of real-time semantics, thus leaving the user the choice of either having fault-tolerance or real-time. In the P_2P space, research has focused on providing real-time streaming capabilities to video and audio systems, while a parallel body of research addressed the use of fault-tolerance in distributed file-sharing systems.

To the best of you knowledge no system, P_2P or otherwise, explores the networking layer as a means to implement more lightweight fault-tolerance support with integrated support for real-time by directly embedding fault-tolerance mechanisms into peer-to-peer systems, taking advantage of their decentralized and resilient nature, while enforcing QoS policies through the use of resource reservation mechanisms.

This work is part of an effort for the development of new middleware technologies at EFACEC[1], to cope with information systems used to manage public, high-speed, transportation networks. 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). Latest production system were deployed in Dublin and Tenerife for public information management and in-vehicle information.

II. Problem Statement

A. Goals

Our goal is to create a system that integrates fault-tolerance mechanisms directly into the P_2P network infrastructure in order to allow transparent lightweight fault-tolerance. The main goals are:

- to ensure that, in the fault-free case, the end-to-end response time for a fault-tolerant version is as close as possible to the response time for its non-fault-tolerant counterpart;
- to support mixed traffic types with specific QoS parameters;
- to ensure that QoS parameters for each type of traffic are met.

B. Non-Goals/Scope

This work does not aim to provide video or audio streaming capabilities through the use of P_2P, but to be able to support fault-tolerance for several types of traffic with different QoS demands.

[1]EFACEC, the largest Portuguese Group in the field of electricity, with a strong presence in systems engineering namely in public transportation systems, employs around 3000 people and has a turnover of almost 500 million euro; it is established in more than 50 countries and exports almost half of its production (c.f. **http://www.efacec.pt**).

C. Challenges

The ultimate challenge is of combining fault-tolerance and real-time support in a single system with a efficient use of hosts and network resources. For that, a efficient fault-tolerance mechanisms must be use, that takes advantage of the properties of P_2P networking, while maintaining the desired QoS goals.

D. Validation Strategy

In order to validate our concepts, we implemented a simulation bench for providing measurements to our prototype. This bench is responsible for injecting faults during simulations, and for retrieving the produced statistical data.

E. Assumptions/Study Model

We have implemented a prototype for a hierarchical peer-to-peer framework, based on the P^3 [15] architecture and in the lines of systems like Gnutella [5] and P-Grid [1]. The peers in the leafs of the hierarchy are sensors that stream data and events to be delivered to clients. To evaluate the prototype we use the following metrics: packet loss, response time, jitter and mesh management overhead. These where evaluated in scenarios that involved one or more peer failures. We aim to show that even with all the fault-tolerance mechanisms running, we are able to introduce only a small amount of overhead while minimizing the latency and jitter in response times.

The remainder of the paper is structured as follows. Section III describes the our case study and the approach to the problem. Section IV describes the metrics we use for evaluation, the experimental setup, and finally the discusses the results we have obtained (subsection IV-I). In section V we do a reflection on our experience while implementing the prototype, and from the results obtained, we speculate about possible solutions to current problems. Section VII ends the paper with the conclusions and references for future work.

III. Approach

A. Case Study

The railroads system are complex in nature due to the multitude of subsystems present, namely public information, in-vehicle information, SCADA and signalling. Our case study focus on the public information subsystem, but in no way is limited to it. This subsystem primarily objective is to manage relevant data between stations and the control center, and vive-versa. The managed data types includes video, audio and events. Video and audio can be streamlined from the control center to a station, or a set of stations, to provide information, such as the next train arrival at a specific track, advertisement and entertainment. But video and audio can also be collected from stations through the use of the CCTV system and streamed to the control center, as part of the security screening processes. Events are triggered when some failure happens, such as hardware failure, network link failure, etc. They can be originate from any point of the network, but their destination is always the control center.

Each station has a computational node that is responsible for managing its equipment, such as speakers, monitors, etc. In turn, a set of stations is managed by a second-tier server, that manages all the aggregation operations, such synchronizing public information announcements. At the top of the infrastructure, is the first-tier server that is controlled by the client node present in the control center, thus fundamentally creating a hierarchical tree based system.

B. Architecture

In this subsection we overview the architecture of the peer-to-peer mesh. The architecture is based on the P^3 framework [15], a hierarchical P_2P mesh.

1) Terminology: There are 3 types of peers: `super-peer`, `sensor-peer`, and `client-peer`. A super-peer is responsible for maintaining the organization of the mesh and for providing access points to the mesh for external peers. A sensor-peer is dedicated to data collection and its transmission to the mesh. Client-peers act as sinks in the network. Requests for data with given QoS parameters are issued from these peers and the mesh routes the relevant data packets from the sensor peers up to the client peers. P^3 networks follow a tree like mesh, in which all the nodes, except the leafs, represent cells. A `cell` is a set of super-peers that collaborate to maintain a portion of the mesh tree.

2) Key Architectural Decisions: P^3 (Figure 1) was chosen because of its natural mapping with our case study III-A, following a tree based mesh network. Every type of traffic has its own dedicated communications channels in order to avoid multiplexing of data.

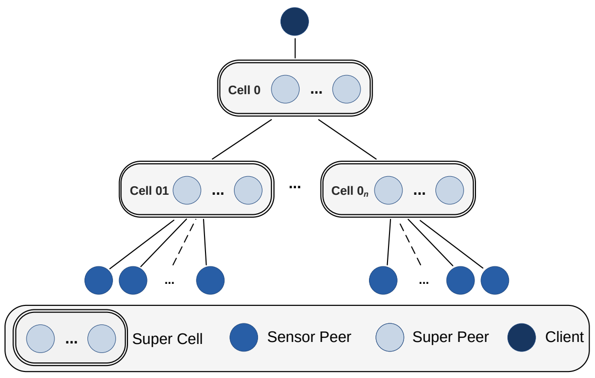


Fig. 1: P3 architecture.

3) Building Blocks: The prototype was built in Java using TCP sockets for intra-cell communication. The discovery service provided in the cell use UDP multicast sockets, while the in-cell fault-tolerance mechanisms are supported by JGroups [2], a reliable group communication framework.

4) Implementation: Each cell is uniquely identified by a `CellID`, represented by a 32-byte string. Each byte represents the order of the cell in a given tree level. For example, the root cell is identified by `'0'`. If `k` is the tree width, the possible siblings of the root cell are {`'01', ..., '0k'`}; in turn the possible siblings of cell `'01'` are {`'011', ..., '01k'`} and so on. Each cell is composed of a set of peers, with unique identifiers, up to a maximum number, a parameter of the simulation. The cells are disjoint. Cell and peer identifiers completely specify the position of a peer in the network (Figure 1).

Each super-peer provides a set of services. The `Mesh` service is the most fundamental service, in the sense that it supports all the remaining services: `Audio`, `Video` and `Events`, and has two sub-services: `Membership` and `Discovery`. The membership service provides the mechanisms for joining and leaving the mesh, and is composed of two layers: the first layer is responsible for the communications within the cell and is supported by the use of a reliable group communication framework (JGroups [2] in our case), while the second layer handles the inter-cell communications, based on TCP/IP sockets. The `Discovery` sub-service handles all the queries and information dissemination in the mesh (including cell routing), and is supported by a lightweight multicast implementation. Both sub-services use the same addressing routine, that is based on the following formulae:

$$tree\_pos(cellid[], span) = \sum_{i=0}^{depth-1} span^i + \sum_{i=0}^{depth} cellid[i] * span^{depth-i}$$

$$tree\_pos\_address(ip, port, cellid[], span) = (ip, port + tree\_pos(cellid[], span))$$

In the above formulae, $cellid[]$ is the argument that contains the string of the cell identifier, $span$ is the argument for specifying the mesh tree span, ip and $port$ specifies the base IP address and port. For example, if we use a tree span of 2, '011' as the cell identifier, '228.1.2.2' and 1000 as the base IP address and port, then the result would be:

$$tree\_pos\_address('228.1.2.2', 1000, '011', 2) = ('228.1.2.2', 1003)$$

The simulator uses ('228.1.2.2',1000) as the base address of `Membership` and ('228.1.2.3',1000) for `Discovery`.

C. Algorithms

In this subsection we overview the basic algorithms used in the construction, management, fault-detection and recovery of the peer-to-peer mesh.

1) Organization: The mesh construction algorithm is depicted in Listing 1. To enter the mesh, a new peer calls the `JoinMesh` procedure (line 1), which then requests a cell to connect itself by making a call to the `RequestCell` procedure (shown in Listing 2, lines 1-10). This procedure multicasts a discovery message that tries to find the peer-to-peer overlay. If it fails, if the root cell is empty and if the joining peer is a super-peer then the returned `CellID` is he root cell identifier. In this case, the `BindToCell` procedure just creates a new cell (the root cell, actually a JGroups group) and the peer becomes the first peer of the overlay. The procedure also starts the network services: `Mesh`, `Event`, `Audio` and `Video` in the peer. If the peer-to-peer overlay is detected, the active peers reply to the join request, `RequestCell` returns an appropriate cell and the peer tries to bind to it. In `BindToCell`, after the peer binds to the cell (joins the corresponding JGroups group), it must request a parent peer from the cell immediately above in the hierarchy (lines 22-28). If this request fails then the join process also fails, otherwise the parent peer is queried for its service endpoints. The joining peer then starts each service using the parent's endpoint information to create connections to the respective services in the parent peer.

Listing 1 Join mesh algorithm

```
 1: procedure JOINMESH
 2:     $cellId \leftarrow RequestCell()$
 3:     if $cellId = \emptyset$ then
 4:         return $\emptyset$
 5:     end if
 6:     $cell \leftarrow$ BindToCell($cellId$)
 7:     if $cell = \emptyset$ then
 8:         return $\emptyset$
 9:     end if
10:     return $cell$
11: end procedure
12: procedure REJOINMESH($cell$)
13:     $cellId \leftarrow cell$.CellID()
14:     $cell \leftarrow$ BindToCell($cellId$)
15:     if $cell = \emptyset$ then
16:         return JoinMesh()
17:     end if
18:     return $cell$
19: end procedure
20: procedure BINDTOCELL($cellId$)
21:     $serviceEndpoints \leftarrow \emptyset$
22:     if $cellId$.isRoot() $\neq True$ then
23:         $parentCellId \leftarrow$ ParentCellId($cellId$)
24:         $parentInfo \leftarrow$ RequestParent($parenCellId$)
25:         if $parentInfo = \emptyset$ then
26:             return $\emptyset$
27:         end if
28:         $serviceEndpoints \leftarrow parentInfo$.GetEndpoints()
29:     end if
30:     for all $service$ in $[Mesh, Video, Audio, Event]$ do
31:         $service$.Start($serviceEndpoints$)
32:     end for
33:     if $status \neq OK$ then
34:         return $\emptyset$
35:     end if
36:     return $cell$
37: end procedure
```

Listing 2 Discovery algorithms

```
 1: procedure REQUESTCELL($peerType$)
 2:     $discovery_{Root} \leftarrow$ GetDiscovery($CellID_{Root}$)
 3:     $requestMsg \leftarrow$ CreateRequestCellMessage()
 4:     $request \leftarrow discovery_{Root}$.SendRequest($requestMsg$)
 5:     if $request = \emptyset$ then
 6:         if $peerType = SuperPeer$ then
 7:             return $CellID_{Root}$
 8:         else
 9:             return $\emptyset$
10:         end if
11:     end if
12:     $request$.WaitForReply()
13:     return $request$.GetReplyMessage()
14: end procedure
15: procedure HANDLEREQUESTCELL($cell, msg$)
16:     if $cell$.CellID().isRoot() $\neq True$ then
17:         return
18:     end if
19:     $reply \leftarrow$ CreateReplyMessage(GetCell($msg$.GetPeerType()))
20:     $cell$.getDiscovery().SendMessage($reply$)
21:     return
22: end procedure
23: procedure REQUESTPARENT($cellDiscovery, peerType$)
24:     $requestMsg \leftarrow$ CreateRequestParentMessage($peerType$)
25:     $request \leftarrow cellDiscovery$.SendRequest($requestMsg$)
26:     if $request = \emptyset$ then
27:         return $\emptyset$
28:     end if
29:     $request$.WaitForReply()
30:     return $request$.GetReplyMessage()
31: end procedure
32: procedure HANDLEREQUESTPARENT($cell, msg$)
33:     if IsPeerAllowed($msg$.getPeerType() then
34:         $reply \leftarrow$ CreateAcceptMessage(getServicesInfo())
35:         $cell$.getDiscovery().SendMessage($reply$)
36:     end if
37: end procedure
```

If the binding cell is not the root cell, the joining process will only be complete when the `Mesh` service startup completes. The startup (and leave) algorithms are depicted in Listing 3. The startup procedure begins with the peer sending the join message to its cell (line 3); if the cell is not the root, then the `Mesh` service creates a connection to the parent's service, while hashing a pending request in the current cell, containing information about the original message and peer (lines 4-8). This information is used to match future acknowledgments to the request, insuring the integrity of the mesh. Afterward, a new message containing the join information is created to be sent to the parent cell (lines 9-10). Each cell must repeat this procedure until the root cell is reached. When the root cell receives the join request and updates its cell, it also replies downwards to the requesting sibling cell. When a peer receives a reply message (line 26), it looks at the pending request hash, retrieves the message and peer information, and replies to the appropriate sibling. This process is repeated recursively until the message reaches the joining peer. The procedure for leaving the mesh follows the same scheme, line 18, the only difference being the type of message being sent.

The `Discovery` interface is depicted in Listing 2. When a peer needs a new cell to bind to, it calls the procedure `RequestCell` (lines 1-14) on the root cell, which in turn sends a cell request message to the $\texttt{discovery}_{Root}$ multicast address. The call will be serviced by any of the super-peers in the cell. If there are no peers in the root cell the procedure returns the root cell identifier. Otherwise, it returns an appropriate place on the mesh tree to position the requesting peer (lines 19-20, procedure `GetCell`). The optimal position for a new peer, depends on the strategy used, but given the tree like architecture for the simulator, it is clear that we should try to occupy the top of the tree first, in order to provide a more resilient infra-structure (in Subsection IV-I we will see that the consequences of a fault depend heavily on its location on the tree mesh). The procedures `RequestParent` is used to query the service endpoint information for the parent of the requesting peer. `HandleRequestParent` is the call-back on the parent side that processes such requests. For now, we do not impose any restrictions on the number of clients each parent has.

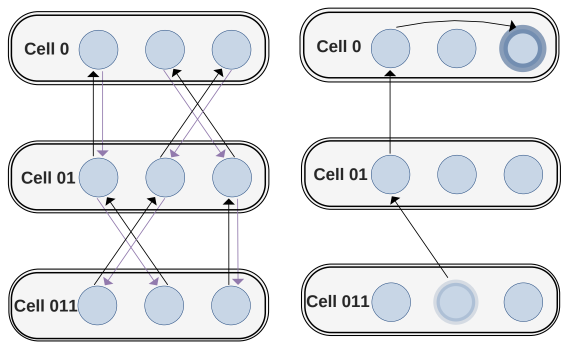
Listing 3 Mesh service membership algorithms

```
 1: procedure MeshServiceStartup($cell, parentEndpoint$)
 2:     $joinMsg$ ←CreateJoinMsg()
 3:     $cell$.SendMessage($joinMsg$)
 4:     if $cell$.IsRoot() $\neq True$ then
 5:         $parent$←BindToParent($parentEndpoint$)
 6:         if $parent = \emptyset$ then
 7:             return $\emptyset$
 8:         end if
 9:         $request$ ← CreateAndStoreRequest($joinMsg$)
10:         $parent$.SendMessage($joinMsg$)
11:         $request$.WaitForReply()
12:         if $request$.Failed() then
13:             return $\emptyset$
14:         end if
15:         return $request$.GetReplyMessage()
16:     end if
17: end procedure
18: procedure MeshServiceLeave($cell, parent$)
19:     $leaveMsg$ ←CreateLeaveMsg()
20:     $cell$.SendMessage(leaveMsg)
21:     if $cell$.IsRoot() $\neq True$ then
22:         CreateAndStoreRequest($leaveMsg$)
23:         $parent$.SendMessage($leaveMsg$)
24:     end if
25: end procedure
26: procedure HandleMeshSvcMsg($cell, parent, peer, msg$)
27:     $cell$.SendMessage(msg)
28:     if $cell$.IsRoot() $\neq True$ then
29:         if $peer \neq parent$ then
30:             CreateAndStoreRequest($msg$)
31:             $parent$.SendMessage($msg$)
32:         else
33:             $request$ ← GetPendingRequest($msg.id$)
34:             if $rRequest = \emptyset$ then
35:                 return
36:             end if
37:             $child$ ← $request$.GetPeer()
38:             $child$.SendMessage($msg$)
39:         end if
40:     else
41:         $replyMsg$ ←ProcessMessage($msg$)
42:         $peer$.sendMessage($replyMsg$)
43:     end if
44: end procedure
```

Routing

The routing mechanism in the peer-to-peer mesh is rather straightforward and is presented in figure 2.

The process is illustrated in Listing 4. When a peer receives

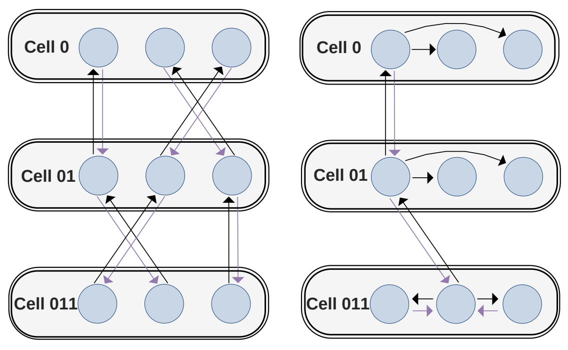


(a) Initial bindings (b) Routing path

Fig. 2: Routing mechanisms.

a message it checks the target peer and cell identifiers. If a direct connection exists to the target peer (lines 4-17) then the message is relayed directly through that link (rather than through the mesh). Otherwise, a check is made to detect whether the message is going upwards or downwards in the hierarchy. In the former case, the message is sent to the parent cell of the current peer (line 8). Otherwise the peer checks whether it is connected directly to the a peer in the target cell (line 10). If so, it sends the message directly through this peer (line 14) otherwise it routes the message through its siblings in the current cell (line 12). If a message is received from a child or from the parent peer, the procedure `HandlePeerMessage` handles the request. If the message belongs to the current peer then the `ProcessMessage` is called (line 21), otherwise the message is re-routed using procedure `SendMessage`. If a message is received via the JGroups framework then the procedure `HandleCellMessage` is called, to check if the message is addressed to the current peer, and if so, process it (line 28), otherwise simply discard the message.

2) Fault-Tolerance: The fault-tolerance mechanisms are illustrated in Figures 3 and 4.



(a) Initial setup (b) Fault Tolerance spreading

Fig. 3: Fault-tolerance spreading.

Figure 3a illustrates the initial setup of the mesh, with the representation of the bindings between peers. When a fault-tolerance message is received by a super-peer in Figure 3b, it starts the spreading process by first propagating the information within its cell, followed with the propagation to the

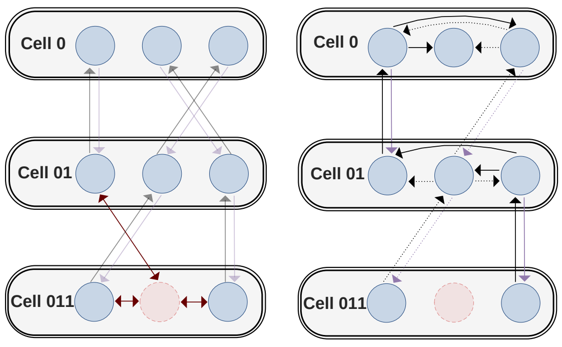
Listing 4 Routing algorithm

```
procedure SendMessage($cell, parent, msg$)
    $peerId \leftarrow msg.peerId$
    $cellId \leftarrow msg.cellId$
    $peer \leftarrow$ GetDirectPeer($peerId, cellId$)
    if $peer = \emptyset$ then
        $direction \leftarrow$CheckDirection($peerId, cellId$)
        if $direction = UP$ then
            $parent$.SendMessage($msg$)
        else
            $peerChild \leftarrow$GetPeerChild($cellId$)
            if $peerChild = \emptyset$ then
                $cell$.RouteMessage($msg$)
            end if
            $peerChild$.SendMessage($msg$)
        end if
    end if
    $peer$.SendMessage($msg$)
end procedure

procedure HandlePeerMessage($msg$)
    if MessageIsOurs($msg$) $= True$ then
        ProcessMessage($msg$)
    else
        SendMessage($msg$)
    end if
end procedure

procedure HandleCellMessage($msg$)
    if MessageIsOurs($msg$) $= True$ then
        ProcessMessage($msg$)
    end if
end procedure
```

immediately upper tree level. This process is recursive and stops when the root cell is reached. The peer that receives the message at the root cell then sends an acknowledgment message down the tree to the source peer.



(a) Failed peer in cell 011 (b) FT recovery

Fig. 4: FT recovery

Figure 4a shows a failure of a peer and consequent detection by the peers in its cell and its parent peer (in the parent cell). The recovery process is depicted in Figure 4b, and consists in the implicit takeover of the pending fault-tolerance messages that originated in the faulty peer. If there are more peers within the faulty peer's cell, then those peers compare their `PeerID` to the faulty peer's `PeerID`, using the *Levenshtein distance*. The peer that has the minimum distance becomes the owner of those pending messages. In the rare case that more than one peer is at the minimum distance we will get duplicates, as both peers retransmit those pending fault-tolerance messages. This is not a problem as the duplicates are detected at a later stage (c.f. failure recovery mechanisms).

The algorithms that implement these mechanisms are shown in Listings 5, 6, 7, and 8.

Listing 5 Fault-Tolerance algorithms

```
 1: procedure SendFTMessage($peerId, cell, service, msg$)
 2:     if not $msg$.IsFaultTolerant() then
 3:         return
 4:     end if
 5:     $ourMsg \leftarrow$ ($peerId$ = $msg$.GetSourcePeerId())
 6:     $sourcePeer \leftarrow msg$.GetSourcePeerId())
 7:     $ftRequest \leftarrow service$.GetPendingRequest($sourcePeer$)
 8:     if $ftRequest = \emptyset$ then
 9:         if $msg$.type $\neq$ FT then
10:             $msg \leftarrow$CreateFTMsg($peerId$,$cell$.GetCellId(),$msg$)
11:         end if
12:         $ftRequest \leftarrow service$.CreateAndStoreRequest($msg$)
13:     end if
14:     $cellStatus \leftarrow cell$.SendMessage($msg$)
15:     if $cellStatus \neq OK$ then
16:         if $ourMsg = True$ then
17:             $ftRequest$.StartTimer()
18:         end if
19:         return
20:     end if
21:     if $cell$.IsRoot() $\neq True$ then
22:         $parent \leftarrow service$.GetParent()
23:         $parentStatus \leftarrow serviceParent$.SendMessage($msg$)
24:         if $ourMsg = True$ then
25:             $ftRequest$.StartTimer()
26:             return
27:         end if
28:     end if
29:     return
30: end procedure

31: procedure HandleFtTimeout($service, ftRequest$)
32:     $service$.SendFtMessage($ftRequest$.GetFTMessage())
33: end procedure
```

The procedure `SendFTMessage`, in Listing 5, is responsible for the upwards propagation of a message while actively replicating it across the way. In line 2, the message is checked to see if it is to be handled by the fault-tolerance mechanism, returning immediately if this is not the case. If a request is not stored to handle this message, then a new request must be created to handle the message. If the message is of type `FT`, then it is owned by another super-peer, otherwise we encapsulate the original message in a `FTMessage`. At this point, the request is created with the corresponding fault-tolerance message (lines 6-13) and the later is sent to the peer's cell (line 14). If this operation fails (line 15), and if the peer is the owner of the message then a timer is set to repeat this process, with an associated timeout handler (lines 31-33). The message is propagated until it reaches the root cell (lines 21-28). If the peer is not located in the root cell, then the message is sent to its parent. If this message fails and the peer is the owner of the message, a timer is set to repeat the process.

The mechanism that manages all the incoming messages for a peer (from the parent peer or child peers) is illustrated in procedure `HandleFTMsg` in Listing 6. It receives as arguments the connection that originated the message (`service`), as well the message itself. Upon reception of a message originating from a child (lines 2-7), the peer tries to propagate

Listing 6 Fault-Tolerance handler for neighbors messages

```
1: procedure HandleFtMsg($cell, peer, service, parent, msg$)
2:     if $peer \neq parent$ then
3:         $parentStatus \leftarrow parent$.SendFTMessage($msg$)
4:         if $parentStatus \neq OK$ then
5:             $replyMsg \leftarrow service$.CreateFTFailureMessage()
6:             $peer$.SendMessage($replyMsg$)
7:         end if
8:     else
9:         $request \leftarrow service$.GetPendingRequest($msg.id$)
10:        if $msg$.GetReply() $= OK$ then
11:            $service$.RemoveStore($msg$)
12:            $cell$.SendMessage($msg$)
13:            if $peer$.GetPeerId()$\neq$ $msg$.GetSourcePeerId() then
14:                $child \leftarrow request$.GetPeer()
15:                $child$.SendMessage($msg$)
16:            end if
17:        else
18:            if $peer$.GetPeerId() $=$ $msg$.GetSourcePeerId() then
19:                $request$.StartTimer()
20:            end if
21:        end if
22:    end if
23: end procedure
```

it to the parent and, if the `SendFTMsg` procedure fails, notifies the child. Next follows the code that handles messages received from the parent (lines 9-22). If the respective request is still pending and the propagation was successful, then the peer removes the request and propagates the message to the other peers in the cell (lines 9-12). If the peer is the not owner of the message, then the message is sent to the child that originated the request (lines 13-16). In case of an unsuccessful reply and if the peer is the owner of the request, a timer is set for request retransmission (lines 18-20).

Listing 7 Fault-Tolerance handler for incoming cell messages

```
1: procedure HandleFtCellMsg($service, msg$)
2:     if $msg.type = FT\_REPLY$ then
3:         $service$.RemoveStore($msg$)
4:     else
5:         $service$.Store($msg$)
6:     end if
7: end procedure
```

The intra-cell handler for the messages is `HandleFTCellMsg`, shown in Listing 7, and it is responsible for processing the messages originating from any sibling on the current cell. These messages are sent and received through the JGroup's framework. In the fault-tolerance context, this handler removes the pending request if the received message is of the `FT_REPLY` type; otherwise we are in the presence of a new fault-tolerance message, and the information is to be stored. These requests are stored to be used in the recovery process in case of a peer failure. Currently, this *active replication* is the only one implemented in the simulator. This process is illustrated in Listing 8.

The procedure `OnFail` present in Listing 8 is called on every peer in the faulty peer's cell, and on its parent. If the parent detects that the cell does have anymore peers, then it assumes the ownership of the faulty peer's pending requests (requests received but for which no acknowledgement reply has been received), otherwise it remains inactive. If there

Listing 8 Fault-Tolerance recovery algorithm

```
1: procedure OnFail($cell, peerID, faultyCellId, faultyPeerId$)
2:     if $cell$.GetCellId() = $faultyCellId$ then
3:         $peerList$ ←GetCellSuperPeers($cellId$)
4:         $recoveryId$ ←MinDistance($faultyPeerId$,$peerList$)
5:         if $peerId$ = $recoveryId$ then
6:             $depList \leftarrow$ GetSubTree($faultyPeerId$)
7:             for all $peer$ in $depList$ do
8:                 $msgList \leftarrow$ StoredFTMessages($peer$)
9:                 for all $ftMsg$ in $msgList$ do
10:                    SendFTMessage($ftMsg$)
11:                end for
12:            end for
13:        end if
14:    else
15:        $cellId \leftarrow cell$.GetCellId()
16:        if $cellId = faultyCellId$.getParentCellId() then
17:            if IsCellEmpty($faultyCellId$) = $True$ then
18:                $depList \leftarrow$ GetSubTree($faultyPeerId$)
19:                for all $peer$ in $depList$ do
20:                    $msgList \leftarrow$ StoredFTMessages($peer$)
21:                    for all $ftMsg$ in $msgList$ do
22:                        SendFTMessage($ftMsg$)
23:                    end for
24:                end for
25:            end if
26:        end if
27:    end if
28: end procedure
```

are other peers in the faulty peer's cell, then all those peers calculate the lexicographic distance between their `PeerId` and the faulty peer's `PeerID`. The peer that has the minimum distance assumes ownership of the pending requests. If two or more peers have the same minimum distance this will result in duplicate messages being sent to the their parent cell. The duplicates are then detected and discarded, thus preserving correctness.

IV. Validation

A. Simulator Bench

The simulator creates a network with n peers and m sensor-peers. The super-peers are grouped in cells that are created according to the rules of the underlying P^3 framework. Table I illustrates the most relevant simulation parameters.

Property	Type	Meaning	Default
FT	Boolean	Turns on/off FT	N/A
TREE_SPAN_DEPTH	Integer	Tree span per DEPTH	N/A
SPS_PER_CELL_DEPTH	Integer	Peers per cell and DEPTH	N/A
SENSORS_PER_CELL	Integer	Max sensors per cell	2
AUDIO_RATE	Integer	Audio frames per second	38
AUDIO_FRAME	Integer	Audio frame size in bytes	418
AUDIO_PACKETS	Integer	Audio packets sent in simulation	9790
VIDEO_RATE	Integer	Video frames per second	24
VIDEO_FRAME	Integer	Video frame size in bytes	4180
VIDEO_PACKETS	Integer	Video packets sent in simulation	6120
EVENTS_RATE	Integer	Events per second	25
EVENTS_PAYLOAD	Integer	Events payload size in bytes	1024
EVENTS_PACKETS	Integer	Event packets sent in simulation	6375

TABLE I: Simulator parameters.

B. Experimental Setup

The simulations were run in a cluster composed of the following nodes:

Name	Processor	Core Count	Memory
$\mathtt{N}_0$	Intel Corei7 920@`2.67Ghz`	4	6Gb
$\mathtt{N}_1$	Intel Core2 Q9450@`2.4Ghz`	4	2Gb
$\mathtt{N}_{\{2,3\}}$	AMD Athlon X2@`1.0Ghz`	2	2Gb
Total Cores	12		

TABLE II: Simulation setup.

The physical infrastructure was based on a 100 Mbit/s Ethernet network with a star topology. The sensor nodes were allocated in the $\mathtt{N}_2$ node, the client was positioned on the $\mathtt{N}_3$ node, and the super-peers assigned to the nodes $\mathtt{N}_0$ and $\mathtt{N}_1$.

C. Metrics for Fault-Tolerance and Real-Time

The following metrics were used to evaluate the fault-tolerance behavior: sensibility to fault location/data loss; fault-detection latency; and availability. In order to assess the behavior of real-time we evaluated the overhead introduced by the fault-tolerance mechanisms. For that we measure the latency and jitter for the packets sent by the sensors to the client (one-way), and the amount of resources used, namely cpu time and memory.

D. Building Blocks

The simulation bench was built using *python* scripting, that is reponsible for: a) launching the simulations run; b) collecting statistical data from all the nodes; and c) processing all the statistical data.

E. Experiments & Fault-Injection Campaign

The experiments were designed to evaluate the performance of all previously stated metrics.

F. Analysis of Fault-Tolerance Experimental Results

1) Sensibility to Fault Location & Data Loss:

2) Fault-Detection Latency:

3) Availability:

G. Analysis of Real-Time Experimental Results

1) Response Time & Jitter:

2) Resource Usage:

3) Nagle's Algorithm Effects on Real-Time:

H. Analysis of Overall Scalability & Performance

1) Scalability:

2) Throughput: The experiments were designed to evaluate the following scenario, in the presence of peer failures:

> *A client sends a request to a sensor-peer for data with QoS constraints. Data, in the form of events or video/audio frames is sent to the client. Eventually some packets are lost. A packet is considered lost if it does not meet the QoS requirements for the particular type of traffic.*

The type of QoS constraints we use in the simulator is very simple: we just check the amount of packets that did not arrive at the client in each simulation run. To test the impact of the fault tolerance support described in Section III we use a P^3 network consisting in a binary tree span with two super-peers per cell, in all the levels of the tree.

We have created four distinct test scenarios to evaluate the fault tolerance mechanisms. The only fault-tolerance strategy used for these experiments was *active replication*. The test scenarios are: **FT_ALL**, uses fault-tolerance for all type of traffic; **FT_EVENT** only uses fault-tolerance for events; **FT_EVENT+VIDEO** uses fault-tolerance for events and video, and; **No_FT** has fault tolerance disabled. In order to assess the impact of different types of faults and at different locations of the tree, we designed the following tests:

Test I	one fault per tree level.
Test II	incremental faults until the total number of peers present in each level is reached.

We used the following relative priorities for each traffic type: video and audio – 1; events – 3; mesh – 10.

I. Analysis of Experimental Results

In all the results presented below, the following color scheme was followed: video appears as yellow line or bar; audio is shown as a blue bar or line; and events are represented by a red bar or line.

Test I allows us to visualize the behavior of the system when a peer failure happens at different levels of the tree.
NO_FT: this is the control scenario, depicted in Fig. 5, for

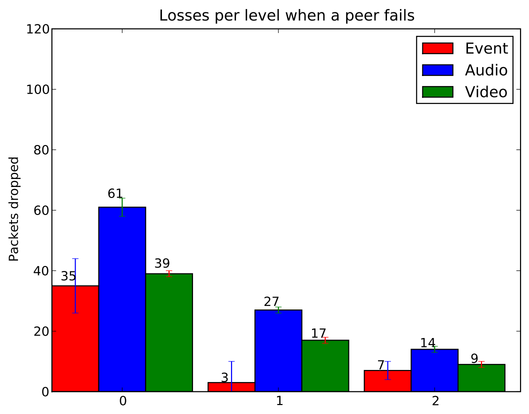


Fig. 5: Results for Test I without FT.

Video latency (ms)	21 ± 2
Audio latency (ms)	39 ± 3
Event latency (ms)	21 ± 3
Slow rebind (ms)	1109 ± 896
Fast rebind (ms)	47 ± 4

TABLE III: Test I latency statistics without FT.

		Overhead over no-FT (%)
Video latency (ms)	24 ± 3	114
Audio latency (ms)	40 ± 2	103
Event latency (ms)	24 ± 2	114
Slow rebind (ms)	1090 ± 888	98
Fast rebind (ms)	52 ± 9	111

TABLE IV: Test I latency statistics with FT for events.

		Overhead over no-FT (%)
Video latency (ms)	39 ± 8	186
Audio latency (ms)	49 ± 2	126
Event latency (ms)	39 ± 2	186
Slow rebind (ms)	1094 ± 893	99
Fast rebind (ms)	77 ± 8	164

TABLE V: Test I latency statistics with FT for events and video.

which no fault-tolerance mechanisms are active. The results were as expected with losses increasing as the faults got nearer of the root level. This is a result of the fact that peers closer to the root handle larger loads of traffic, and therefore a failure here causes more disruption. The video and audio traffic behave consistently with this trend, while the event traffic does not quite. The unusual behavior of event losses is due to the fact that they are intrinsically random in nature, in contrast to the time constant streams of video and audio. Table III shows the statistical information about latency and jitter (for client to sensor-peer communication), for all the traffic types, as well the information for the rebind mechanisms. There are two types of rebinding: ***slow rebind*** and ***fast rebind***. The slow rebind happens when a peer, in case of its parent failure, must execute the full join procedure (described in the previous chapter), while the fast rebind happens when a peer successfully rebinds to another parent peer (in the same cell of the failing peer), using previous retrieved discovery information (from the time of the first join).

FT_EVENT. Fig. 6 shows the results for the runs with the fault-tolerance mechanisms active for events. This fact completely avoids the losses of events when the failing peers are in levels 0 and 1. Sensor-peers do not have any type of memory to preserve packets until they are safely delivered. This leads to packet losses when their parent peer fails, forcing the sensor-peer to rebind to another parent. Until the rebind process finishes, all the packets that should be sent to the mesh are discarded because the sensor does not have a valid parent to route the traffic.

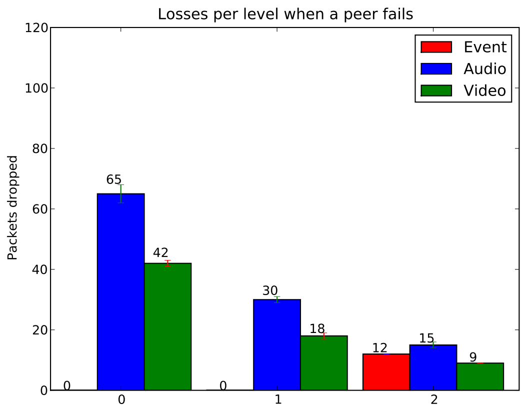


Fig. 6: Results for Test I with FT for events.

Table IV has the statistical information for this test. The presence of fault-tolerance introduces an increase of latency in all types of traffic, a consequence of the time spent in executing the fault-tolerance semantics.

FT_EVENT+VIDEO. Fig. 7 shows how the simulator copes with two types of traffic being supported by the fault-tolerance infrastructure. The increase in audio packet losses reflects the cost of maintaining both event and video traffic with fault-tolerance support, as more resources are needed, such CPU time and network bandwidth. As in the above tests, in level 2, the fault-tolerance mechanisms are not able to avoid packet loss because of the sensor's rebind time. Table V shows that

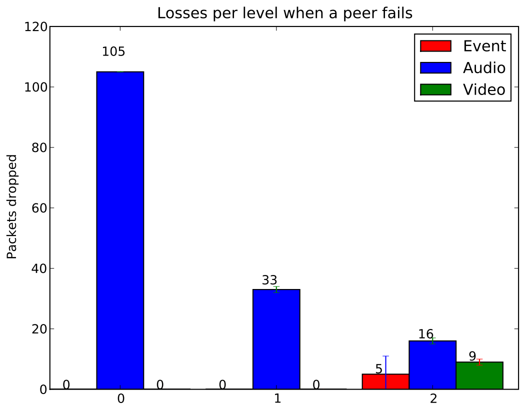


Fig. 7: Results for Test I with FT in event and video traffic.

all traffic types suffer an increase in their latency time, as explained before, the competition for computational resources and network traffic explains this behavior.

FT_ALL. Fig. 8 shows the results for the runs with all the traffic types (video, audio and events) being supported by the fault-tolerance mechanisms. The simulator is able to successfully support all the three traffic types without losses when the failing peer is in levels 0 and 1. For level 2, the behavior of the simulator is in line with the previous results, where the rebind time of the sensor-peer dictates the amount of packet loss for any given traffic type. Table VI shows that the cost of maintaining all the three types of traffic with fault-tolerance comes with the cost of increased latency.

Tables III to VI show the response time and jitter associated

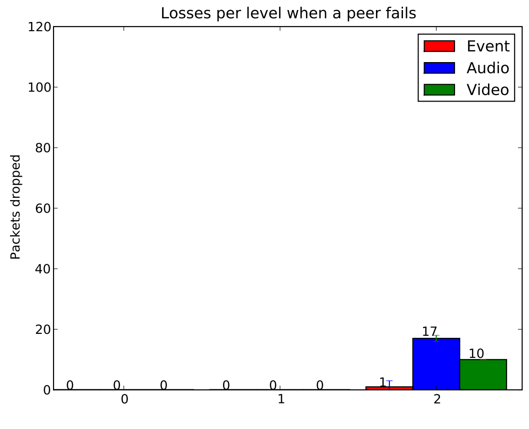


Fig. 8: Results for Test I with FT for all traffic types.

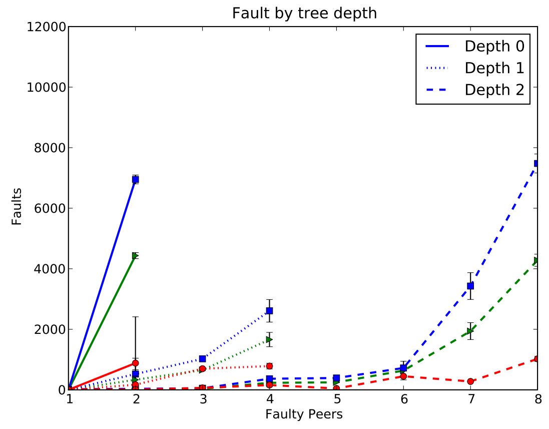


Fig. 9: Results for Test II with FT for all traffic types.

		Overhead over no-FT (%)
Video latency (ms)	49 ± 4	233
Audio latency (ms)	64 ± 3	164
Event latency (ms)	49 ± 3	233
Slow rebind (ms)	1088 ± 885	98
Fast rebind (ms)	57 ± 16	121

TABLE VI: Test I latency statistics with FT for all traffic types.

with each type of traffic as the fault-tolerance mechanisms are activated, as well as the overhead involved. The systematically higher latencies observed for audio traffic is due to the fact that it has a higher frame rate (*c.f.* Table I). The overhead is largely dominated by active replication of data in the cells that is accomplished using JGroups. The results show that the overhead is small for traffic such as events but that it rises in proportion to the total number of packets per second being sent by the sensor-peers.

Test II allows us to analyze the behavior of the system with the increasing number of failures while all the traffic types are being supported by the fault-tolerance mechanisms. This is in fact a worst case scenario, used to stress the system and evaluate its performance. The faults can affect a single peer or an entire cell, depending on the target faults, i.e. if we have a cell with four peers, then the fourth run in the tests that represent the failure of all the four peers in the cell, can be either composed of four independent single peer failures or from an entire cell crash (given that the cell has four peers). Fig. 9 shows the results for running test II on the same P^3 binary tree. When the root cell crashes it results in a large increase of packet losses, even with FT active. This is because the recovery time is dominated by the reconstruction of the mesh tree. As the failures progress to the lower levels, the FT mechanisms help minimize the packet loss. In level 1, the packet losses remain fairly low due to the faster recovery time. As mentioned earlier, the FT helps minimize the losses, that rise modestly when faults extend to all of the cells in the level. In level 2, the overall packet loss rise modestly up to 6 (out of 8 possible) failures of super-peers, and only then it experiences a step rise. This is a simulation artifact due to the restriction that there cannot be more than 2 sensor-peers per cell and the root cell does not accept connections from sensor-peers. The number of sensor-peers per cell is a simulation parameter. In the worst case scenario, when the 6th super-peer fails at level 2 we may reach a situation in which the cells at level 1 and the remaining cells at level 2 are at their full capacity (2 sensor-peers per cell). After the 6th super-peer in level 2 fails, the number of available cells is insufficient to support all the sensor-peers and therefore part of the traffic generated by the sensor-peers is lost.

Level 0				
#Peer Faults	**1**	**2**		
Video latency (ms)	55± 1	443 ± 18		
Audio latency (ms)	72± 2	448 ± 17		
Event latency (ms)	55± 2	443 ± 17		
Level 1				
#Peer Faults	**1**	**2**	**3**	**4**
Video latency (ms)	53 ± 3	89 ± 29	103 ± 1	116 ± 4
Audio latency (ms)	69 ± 3	105 ± 30	118 ± 1	144 ± 14
Event latency (ms)	53 ± 3	89 ± 30	103 ± 1	116 ± 14
Level 2				
#Peer Faults	**1**	**2**	**3**	**4**
Video latency (ms)	47 ± 1	49 ± 2	46 ± 2	44 ± 2
Audio latency (ms)	63 ± 1	65 ± 2	59 ± 2	61 ± 2
Event latency (ms)	47 ± 1	49 ± 2	46 ± 2	44 ± 2
#Peer Faults	**5**	**6**	**7**	**8**
Video latency (ms)	45 ± 3	43 ± 3	48 ± 6	46± 4
Audio latency (ms)	59 ± 2	59 ± 2	62 ± 5	62± 7
Event latency (ms)	45 ± 2	43 ± 2	48 ± 5	46± 7

TABLE VII: Test II latency statistics with FT for all traffic types.

Table VII shows the response time and jitter for each type of traffic for each level and for increasing super-peer failures. At level 0 the eightfold increase in response time from one to two peer failures is due to the overhead of re-building the P^3 mesh after the root cell crashes. At level 1, the increase in response time and jitter is more modest (only a twofold increase for 4 peer failures). Finally, at level 2 the response time remains fairly constant, and the jitter low, because we are measuring only the traffic that gets up to the client and not taking into account that some sensor-peers fail to send data at all (*c.f.* Fig. 9).

V. Insights & Lessons Learned

From the experience gained throughout the implementation of our prototype, we seems clear the approach taken by all the systems to date, including ours, of using group communication, seems a hurdle for the implementation of a fully fledge real-time fault-tolerant system. We speculate that with the implementation of low-levels resource reservation mechanisms that are able to provide QoS guarantees, and by using strategic point-to-point communications (TCP, UDP or RDS), we will be able to provide a more strict real-time guarantees.

VI. Related Work

Research in fault-tolerance and real-time spaces have focused mostly on CORBA and its siblings, for which specifications have been proposed for Fault-Tolerant [12], [13] and Real-Time [7], [17], [14] support, but whose integration 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. 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 [11] 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.

Research in P_2P space has focused on architecture, protocols and algorithms that addresses fault-tolerance, QoS and some aspects of real-time, on distributed file-sharing systems [10], [4], [3], [9]. Moreover, there are several frameworks that provide system developers with components and patterns to implement custom peer-to-peer systems [8], [15], [20] and some examples of peer-to-peer middleware infra-structures [16], [19], [6].

VII. Conclusions and Future Work

In this paper we addressed the problem of providing real-time fault-tolerance for peer-to-peer networks. We presented a simulation environment for hierarchical peer-to-peer networks based on P^3 [15] and used it to explore the trade-offs between fault-tolerance mechanism overheads, mesh management overheads and QoS measures such as response time, jitter and packet loss. The simulation environment was loaded with data streaming from sensor peers and perturbed with random peer crash failures.

The experiments show that fault-tolerance mechanisms can be implemented directly over the peer-to-peer infra-structure, taking advantage of the topological arrangement of the network. The overhead in response time introduced is modest, as is the jitter, except for the very extreme case where full FT was specified for all the data streams and for all sensors. This is so despite the less than ideal use of JGroups for ensuring mesh consistency and for strong replica consistency in the active fault-tolerance mechanism we used. JGroups is clearly a bottleneck in the simulator (which is somewhat to be expected since it was not developed with such extreme target applications in mind), as is shown by the large overheads when peers have to resort to the mesh to re-bind after all the peers in their parent cell fail. The overhead in response time when FT is activated is due to the distribution of packet copies between the peers of a cell using JGroups. The active FT strategy used is quite resource hungry and contributes also to the observed overhead.

We are implementing different FT mechanisms such as passive FT and hybrid passive/active FT to evaluate their performance under similar load conditions. Our medium term goal is to provide a proof of concept for real-time fault-tolerant peer-to-peer networks that may be used as the networking layer for high-performance middleware, allowing for transparent and efficient support for fault-tolerant, QoS computing.

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).

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] 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.

[4] 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.

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

[6] J. Gerke and B. Stiller. A Service-Oriented Peer-to-Peer Middleware. In *KiVS*, pages 3–15, 2005.

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

[8] L. Gong. *Project JXTA: A Technical Overview*. Sun Microsystems, Apr. 2001.

[9] 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.

[10] 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.

[11] 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. *Concurr. Comput. : Pract. Exper.*, 17(12):1527–1545, 2005.

[12] 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.

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

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

[15] 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.

[16] A. Rowstron, A. Kermarrec, M., and P. Druschel. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[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] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM Special Interrest Group on Data Communication Conference (SIGCOMM'01)*, volume 31, 4 of *Computer Communication Review*, pages 149–160. ACM Press, August 2001.

[20] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 2003.