RPC-V: Toward Fault-Tolerant RPC for Internet Connected Desktop Grids with Volatile Nodes

S. Djilali, T. Herault, Oleg Lodygensky, T. Morlier, Gilles Fedak, F. Cappello

▶ **To cite this version:**

S. Djilali, T. Herault, Oleg Lodygensky, T. Morlier, Gilles Fedak, et al.. RPC-V: Toward Fault-Tolerant RPC for Internet Connected Desktop Grids with Volatile Nodes. SuperComputing 2004 (SC2004), Nov 2004, Pittsburgh, United States. pp.39-39, 10.1109/SC.2004.51 . in2p3-00457039

HAL Id: in2p3-00457039
https://in2p3.hal.science/in2p3-00457039v1

Submitted on 16 Feb 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RPC-V: Toward Fault-Tolerant RPC for Internet Connected Desktop Grids with Volatile Nodes

Samir Djilali, Thomas Hérault,
Oleg Lodygensky, Tangui Morlier, Gilles Fedak and Franck Cappello
INRIA, LRI, Université de Paris Sud, Orsay, France
email: {djilali, herault, lodygens, tmorlier, fedak, fci}@lri.fr

Abstract

RPC is one of the programming models envisioned for the Grid. In Internet connected Large Scale Grids such as Desktop Grids, nodes and networks failures are not rare events. This paper provides several contributions, examining the feasibility and limits of fault-tolerant RPC on these platforms.

First, we characterize these Grids from their fundamental features and demonstrate that their applications scope should be safely restricted to stateless services.

Second, we present a new fault-tolerant RPC protocol associating an original combination of three-tier architecture, passive replication and message logging. We describe RPC-V, an implementation of the proposed protocol within the XtremWeb Desktop Grid middleware.

Third, we evaluate the performance of RPC-V and the impact of faults on the execution time, using a real life application on a Desktop Grid testbed assembling nodes in France and USA. We demonstrate that RPC-V allows the applications to continue their execution while key system components fail.

1 Introduction

Large scale distributed systems like Desktop Grids, Global and P2P computing systems, harness large set of Internet connected resources. Among the large number of issues raised by these systems (security, privacy, coordination, scheduling, load balancing, etc.), two are directly connected to application programming: programming paradigms/models and fault tolerance.

Several message passing models are currently investigated for the Grid. A significant research effort focuses on RPC. GridRPC [30], Ninf[23], Netsolve[9], GridSolve [7] and OmniRPC [29] are examples of RPC specifications and implementations. Very few investigate the issue of fault tolerance. Only a study on Netsolve [25] reports the fault tolerance of one part of the environment: the servers. In this paper we investigate fault tolerance in a wider perspective, considering that all component types of the system may fail potentially and simultaneously.

There are many studies about fault tolerance in RPC like environments. The related work section presents a selection of them. Most of the previous works have investigated the approach of automatic/transparent fault tolerance in the context where the system exhibits the properties of a pseudo-synchronous network. Thus, they basically rely on replication techniques, using reliable group communication primitives. Internet connected Desktop Grids raise two additional issues: scalability and less controllable behavior that may preclude such techniques.

This paper studies transparent and automatic fault-tolerant RPC in the context of large scale Grids, considering an actual implementation. The second section investigates the application scope according to Internet connected Desktop Grid characteristics. The related work section shows that none of the previous works has studied the same problem and that no existing solution fits the Desktop Grid context. Section 4 describes the protocol and the implementation of RPC-V. In section 5, we examine the properties of RPC-V in terms of performance and fault tolerance.

2 Problem Statement

The applications scope depends on the programming paradigm concepts and the type of services (stateless and stateful) that are guaranteed to work on the considered platform. In the rest of the paper we will focus on RPC programming style on Grid. For the sake of simplicity, we call a "service" the function executed by a RPC server in response to a client RPC call, assuming that servers and clients are connected by the Grid.

2.1 High Level Concepts of RPC for the Grids

In order to address fundamental issues, we restrict our study to some key concepts of RPC for large scale Grids. For example, we limit the scope of our study to simple scheduling and data communication modes. These concepts are synthesized from an analysis of existing environments and proposed specifications: Netsolve [9], Gridsolve [7], Ninf[23], OmniRPC [29], GridRPC [30], NES [8], NEOS [15] and RCS [2].

API. Most RPC environments for the Grid provide very similar programming API, allowing the programmer to call, from the client program, library functions or applications on the server side. Most environments provide blocking and non blocking RPC calls. Consecutive non blocking RPCs lead to concurrent RPC executions at the server side.

Virtualization. The Grids are characterized by dynamicity and heterogeneity of their components, leading to the necessity of a virtualization layer between the application launching RPC calls and the execution servers. Generally, the virtualization layer is implemented by a middle tier between the clients and the servers called the agent in Netsolve, resource manager in NES, server in NEOS and Globus MDS in Ninf.

Basic Fault Management Policies. To cope with server crash or disconnection, different fault tolerance schemes could be considered. In Netsolve, GridSolve and NES, the agent (called resource manager in NES) detects servers failures and re-launches RPC execution without user program intervention. In Ninf and OmniRPC, the programmer should anticipate the situation and should add in its program fault detection and recovery actions. In this paper, we consider the first case.

Scheduling. RPC environments for the Grid use a scheduler, responsible for finding and allocating the most appropriate resources for a given RPC call. For this study, we only consider allocating RPCs on servers individually and independently. A typical mechanism implementing this functionality is the match maker of Condor [27]. As discussed in [10], more complex scheduling heuristics can be implemented at the client side or on top of the Grid infrastructure [11].

Data (RPC Parameters and Results) Communication Mode. We only consider the synchronous communication mode, where parameters are transmitted along with the RPC call, capturing a) classical data transmission where arguments/result are marshaled into a serialization format and b) file transport where a file or a directory is compressed into an archive file.

2.2 The Challenge of Large Scale.

Fundamentally, the characteristics of Internet connected Desktop Grids derive from the combination of best-effort networks and infrastructures gathering a large number of weakly controlled and volatile computing nodes.

Volatility. The size of large scale computing infrastructure makes the node disconnection probability not a rare event. The origin of disconnection can be hazardous (a node shutdown, punctual high response time or crash, a communication timeout due to long network stall or high congestion), or due to management policy restricting node and network utilization for Grid applications only on idle time. Even though progresses may be done in volatility prediction, it is likely that Internet connected Desktop Grids will always suffer of a certain level of unpredictable failures.

No Stable Component. The unpredictability of component volatility precludes to consider the existence of stable components in the system. Thus fault tolerance mechanisms such as fault detectors and recovery techniques should be used on every component of the system. The potentiality of client failures immediately raises the issue of selecting a policy about ongoing RPC calls. In this paper, we consider client disconnection as a normal event, assuming that users may use mobile terminals. Thus we let the execution continue on the server side.

Intermittent Crashes. One element increases the volatility issue: components may leave the system for any period of time without prior notification. For example, nodes interrupted abruptly and restarting from a checkpoint image (like a suspend on disk) may not have time to announce their disconnection, and may restart in a state inconsistent with the rest of the system. More generally, due to the lack of control of the participating nodes, it is impossible to guarantee that the nodes will behave as expected during disconnection and reconnection.

Internet. The Internet is considered as a best-effort network. Its wide performance fluctuations can lead to incorrect fault detection. Usually, fault detectors in fault-tolerant distributed systems provide each system component with a list of suspected dead components. Wrong detections raise two issues: wrong positives (dead components are not suspected), and wrong negatives (alive components are suspected). Some known techniques can be used to limit the wrong positives on the Internet. Wrong negatives are due to unbound message delays, and may not be avoided [12].

Connection-less Interactions. In Internet connected Desktop Grids, it is likely that many clients will be connected to a server at a given time. Reciprocally, it is likely that many servers will be connected to a client launching a large number of non blocking RPCs. Because of system basic settings, the number of simultaneous open connections is bounded. As a consequence, many large scale systems[9, 1, 14] use connection-less interaction protocols: for any interaction with other system components, a connection is opened before the communication and closed immediately after. This technique precludes the use of connection break as a fault detector and implies the use of a "heart beat" signal to detect anomalies.

Changes of the System Size. In large scale Grids, a large portion of the participating nodes connect and disconnect the system from their own initiative, involving huge variations of the system size. The turnover is so high that even nodes participating to the total count are frequently changing [28]. In such system, the notion of majority may have no value since it is likely that a large portion of nodes participating to a decision may not be up when the decision will apply.

2.3 Analysis: stateless or stateful sercices?

Even if we consider the Internet as a pseudo-synchronous network, a) **intermittent crashes**, or b) **connection-less interactions** associated with congestion, introduce unbound delay on message transmission. Following these assumptions, we conservatively characterize large scale Grids connected by Internet as asynchronous distributed systems. Note that some authors are also considering the Internet as fully-asynchronous network [5].

In addition, **volatility** implies that crashes may be permanent. Since there is **No stable component**, the article [20] demonstrates that consensus is impossible in a system characterized by an asynchronous network with at least one failure i.e. where there is no bound on message delay.

A commonly proposed approach to circumvent this problem is to use failure detectors. In [13], the authors define the minimal requirements on failure detectors to solve the consensus problem. These two requirements should hold for a "sufficiently long" period of time for the algorithm to achieve its goal. Unfortunately, the dynamicity of Internet connected Desktop Grids is so high that it is impossible to estimate a period of time, relative to the application execution time, for which the two requirements hold. Furthermore, this approach assumes that a majority of processes are correct, but **changes of the system size** turns majority into a fuzzy notion.

To implement a fault-tolerant stateful services, servers have to agree on a global order of RPC events which is a form of consensus. In [3], the author shows that agreement protocols, even on pseudo-synchronous networks are very slow. In conjunction with the **changes of the system size**, agreement may not be meaningful.

The overall conclusion of this section (and first contribution of the paper) is the following: according to the current knowledge of Internet and Desktop Grid resources behavior, the application scope of Internet connected Desktop Grid should be conservatively restricted to applications calling stateless services and at-least-once semantics. There are too many uncertainties to guarantee the correct execution of stateful services with exactly once semantics.

3 Related Work

We present previous works in several domains: a) Grid, b) large scale distributed systems and c) RPC.

3.1 Fault Tolerance for the Grid

One of the first paper discussing fault tolerance for the Grid [31] proposes, implements and tests an unreliable fault detector service. This paper demonstrates that wrong fault suspicion (wrong positive) occurs even in a Grid with few nodes

In [33], the authors propose a three-tier fault-tolerant architecture associated with replicated executions on the server side. Several coordinators work as middle tiers and vote trying to reach a consensus on the result of replicated jobs. Obviously, this approach is limited to synchronous networks not featuring the issues discussed in the previous section.

GridRPC [30] is a proposal to standardize a remote procedure call (RPC) mechanism for Grid computing, but it does not encompass fault tolerance mechanisms.

Ninf [23] and Ninf-G employ a client-server model but do not provide fault detection nor recovery. Instead, Ninf-G assumes that the back-end queuing system takes this responsibility.

RCS [2] provides an interface to a variety of numerical linear algebra libraries on UNIX platforms but does not implement any fault tolerance mechanism.

NetSolve [25] uses a client-agent-server paradigm and provides two levels of fault tolerance for the servers: a) inter-server fault tolerance moves the computation to another server when the failure of a server is detected by an agent, b) usual techniques of coordinated checkpointing and rollback recovery ensure intra-server fault tolerance. Agent and client fault tolerance is not supported.

In [16], we have developed and tested a RPC environment for P2P systems. However, we restricted our fault tolerance study to server faults, assuming that the other system components are stable.

Legion [24] is an object-based meta-systems designed for large scale distributed systems. Fault tolerance is based on the reflective graph and event model (RGE). This approach allows implementing a large variety of automatic fault-tolerant protocols: coordinated checkpoint, pessimistic message logging, passive replication. However, all implemented protocols assume either a reliable network, reliable storage or some reliable components.

Thus none of the previous works in RPC for the Grid has addressed the issue of full fault tolerance for all system components.

3.2 Large Scale Fault Tolerance

The main problem introduced by the large scale with respect to fault tolerance is the impossibility of consensus described in the previous section. A solution to address this problem is to use probabilistic protocols [26] [21]. However, these protocols can not eliminate some executions, yet unlikely, where the result does not match the consensus.

In [34], the authors describe the benefit of epidemic communication primitives in the context of asynchronous networks. This kind of protocols, inspired by work on large scale database, use reconciliation techniques between neighbors to implement, in a distributed, asynchronous and probabilistic way, the convergence of the system toward a state or a decision. In practice, at large scale, the time required for the convergence of the full system is far higher than the time between two system changes [6]. Thus such protocol could be interesting for establishing agreement within a certain horizon but not at the full system size.

3.3 Fault-tolerant RPC

Fault tolerance in RPC can be addressed at different levels of the software stack.

In [19] the authors propose a replication scheme at the TCP/IP level assuming primary and backup servers. The backup servers use a leader/follower consistency protocol to keep their state consistent with the primary. The protocol relies on the leader election which is impossible in the context of asynchronous network.

In [32] a pool of similar servers, possibly geographically distributed across the Internet, cooperate in sustaining a service by migrating the client connections within the pool. The migration mechanism ensures that the new server resumes the service while preserving the exactly-once delivery semantic across migration, thus assuming a synchronous property of Internet, which is likely to be erroneous at large scale as discussed in previous section.

Fault tolerance has been deeply studied for object oriented distributed systems and especially in the context of the Corba middleware. FT-Corba [5] achieves fault tolerance management by object replication, fault detection and recovery. Recovery is done by electing a new primary object if the current one is suspected to be faulty. In this case too, election means the capability to establish a consensus, which is impossible in our context. One elegant proposal for FT-Corba implementation relies on a three-tier architecture, making the middle-tier the corner stone of the fault tolerance protocol [4]. This protocol partially releases the requirement of network synchrony requiring that the middle tier replicas are deployed on a synchronous system.

In this domain too, there is no previous works considering the context of RPC execution at large scale on asynchronous networks.

According to our knowledge, no research result has been published concerning the problem statement described in the previous section.

4 RPC-V Protocol and Implementation

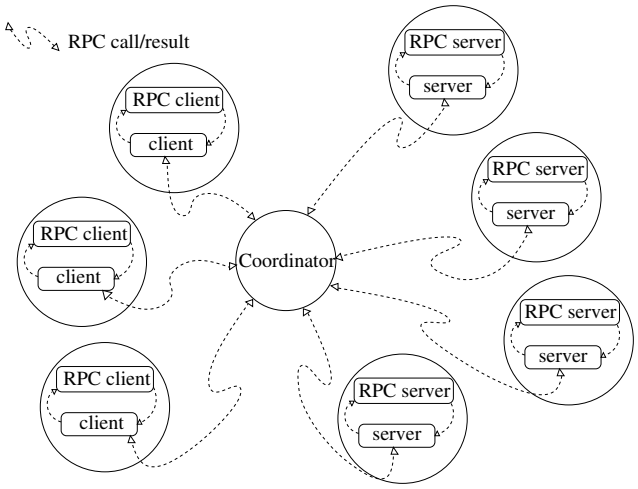

Figure 1. RPC-V General Architecture

RPC-V is an automatic and transparent fault-tolerant environment for RPC programming on Internet connected Desktop Grids. The main features of RPC-V (API, virtualization, basic fault management policies, scheduling and data communication modes) have been designed according to the context of large scale Grids, described in section 2.

It tolerates any fault combination of its system components. Even if Internet connected Desktop Grids limit the application scope to the one calling stateless services, the evaluation section will show that many applications can take benefit of RPC-V.

RPC-V design follows a new fault tolerant architecture based on 1) a three-tier architecture (clients, Coordinator,

servers) described in figure 1 2) message logging on all system components 3) fault detector on all components and 4) passive replication of the coordinators. Despite all these four elements are well known, their combination is original and leads to a robust system as demonstrated in evaluation section.

4.1 Fault-Tolerant Protocol

The fault-tolerant protocol is defined from a fault model, fault detection mechanisms and the actions of every component.

Coordinators communicate together, according to a topology updated on fault suspicion. In the rest of the paper, we denote by Coordinator, the coordination service and by coordinators the components implementing this service.

Each client and server has a preferred coordinator, with which it communicates. This coordinator may not be the same for all the components.

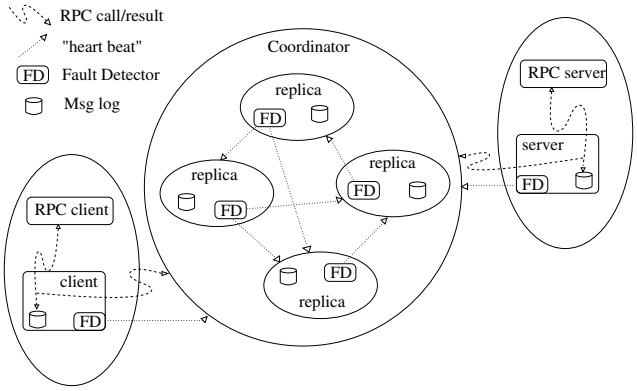

Figure 2. RPC-V Fault-Tolerant Protocol

Model of Faults. Faults can occur at any time on any component (potentially on all components simultaneously), and can be intermittent or permanent. Every restarting component restarts from the beginning of its execution or from its last local state (which can be implemented using uncoordinated checkpointing techniques). Obviously, for the application to progress, at least one client has to submit RPCs, one coordinator has to forward them, one server has to execute submitted RPCs, and the coordinator should be able to communicate with the two other components.

Fault Detectors. The core of the fault-tolerant protocol is the fault detector. As we assume an asynchronous network, the fault detection can only be used for suspecting a component failure. To avoid confusion, in the following sections, we use the term fault suspicion instead of fault detection. As described in figure 2, users ensure the fault suspicion of clients; every component of the system implements fault suspicion of the coordinators; the coordinators implement the fault suspicion of the servers. Thus every component of the system uses at least one fault detector and can be suspected by at least another component.

Preventive Actions. In addition to the component interactions described above, every component performs the following preventive actions:

It locally logs every sent message (sender based message logging). For each communication, components synchronize their local state from these logs. This synchronization is useful to ensure: 1) client applications to rollback their execution to the point exactly following the last RPC call registered on the Coordinator, and 2) servers to re-execute RPCs if their results are not accessible anymore on coordinators.

Additionally, when two coordinators communicate, they synchronize their states, using a passive replication algorithm. Coordinators may have different views of the system due to the network asynchrony.

Actions on Suspicion. When an user suspects a client application failure, it simply re-launches the application, on the same node, or on another one. This feature allows the user to disconnect itself from the system while the computation is still in progress. On reconnection, the client and coordinator synchronize their state from their local logs. This is the responsibility of the user to select the appropriate client and manage the client replicas.

When a client or a server suspects its preferred coordinator, it selects another one, contacts it and they synchronize their states from their local logs.

When a coordinator suspects a server failure, it schedules new instances of all RPC calls forwarded to the suspect, on a new or unsuspected servers. This implements the ”on suspicion” replication strategy.

When a coordinator suspects one of its neighbor coordinators in the topology, it computes a new topology in order to stay in the same connected component.

Note that due to system asynchrony, all components may wrongly suspect all other ones, leading to partition the system. Our protocol guarantees that the client application progresses as long as there is a path between a client and a server (progress condition).

4.2 Implementation

This paper focuses on the protocol and its implementation. We give here a rapid view of the programming API and security approach.

The RPC-V API is compliant with GridRPC [30] except the functions for Remote Function Handle Management that are absent of the RPC-V API. The coordinator virtualization and forwarding avoid the need of function handle management at the client side (the client never connects to the server directly). Instead, this is handled by the coordinator. As a consequence, any client application written following the GridRPC API can be executed on RPC-V. Note that GridRPC API is not transparent for the programmer: application-side invocations do not appear to be local procedure calls.

We have implemented RPC-V protocol on top of the XtremWeb [18] Desktop Grid middleware as a proof of concept. XtremWeb already provides the client, coordinator and server basic mechanisms. XtremWeb is implemented with popular technologies such as Java and MySQL. In XtremWeb, the client submits jobs on the coordinator, which are translated as tasks (instances of jobs) and forwarded to the server (known as the worker in XtremWeb). Jobs in XtremWeb are very close to remote execution calls and encompass command line and an optional directory archive (the called executable is transferred automatically on the server side if necessary).

Security on the Grid means dealing with authentication, authorization, integrity and privacy between clients and servers. In XtremWeb, authentication is done by certificate, integrity is ensured by Sandboxing executions at the server side, privacy can be forced by encrypted communications. Authorisation is checked dynamically by the sandbox and can be done on a per user or community basis. In addition, chains or trees of RPC calls raise the issue of delegation of trust. Currently, delegation is not managed in XtremWeb but it could be implemented using approaches like proxy certificates [22].

The following paragraphs describe the integration of RPC-V protocol in XtremWeb.

Fault Detector. To limit the message traffic and coordinator complexity, we implement the fault detector for coordinators and servers by a ”heart beat” signal sent periodically. When an ”heart beat” signal is timed out, we assume (maybe wrongly) a failure, whatever is the reason: either a crash, a network failure or an intermittent congestion. The ”heart beat” frequency is adjusted considering the tread-off between Coordinator reactivity and congestion.

Coordinator Topology and Preferred Coordinator. We provide all components of the system with a finite list of known coordinators. This list has to be loaded for a first time and updated frequently as it evolves according to fault suspicions. All components download the same list at system initialization from known repositories (web servers, DNS, mail communicated messages, etc...). The list is updated locally from system fault suspicions and merged periodically, at ”heart beat” signal receptions. It is also updated periodically based on user information by contacting the known repositories.

Communication Protocol. Connection-less communication leads to design a system where node disconnections are not faults but features of the system, allowing mobile clients and off-line computing at server side.

The communications between (i) clients and coordinators, and between (ii) servers and coordinators are asymmetric and connection-less. Clients and servers initiate all communications to the coordinators and disconnect between two communications. The coordinators only reply to clients and servers requests.

Synchronization. Synchronization with a coordinator determines received and lost messages, which are resent. The synchronization implementation depends on each component local information. For a client, all messages are tagged with a timestamp (all client RPC submissions are associated with a unique counter value), which is compared to the maximum timestamp known for this client by the coordinator. Between two coordinators, the synchronization exchanges maximum timestamps for all known clients. Since servers may have non-contiguous timestamps for a given client, the synchronization is more complicated, involving a peer-wise comparison of logs.

Managing Message Logs. The garbage collection is a fundamental mechanism associated with message logging. Since logging capacities are bounded, we should decide whether flushing some logs, that may be potentially useful for avoiding re-executions, or stopping computations, reducing the system resource utilization. The garbage collection is distributed among all the components and can be triggered locally according to some conditions, or explicitly by the user.

Any client RPC call execution in the system is identified by: the user unique ID, a session unique ID and a RPC unique ID. A session corresponds to the logging of the user into the system. The session ends on logout. Any instance of the client program may connect the Coordinator with different IP and retrieve results and RPC status using the unique IDs. Note that theses IDs are used only to retrieve information about a RPC call. The security is not implemented using these ID but by several mechanisms as discussed at the beginning of this section.

The client submits RPC by sending function identifier and parameters, to the coordinator. Like most Grid RPC implementations, RPC submission may be non-blocking (asynchronous). The client collects the RPC results by pulling the coordinator periodically. RPC submission and

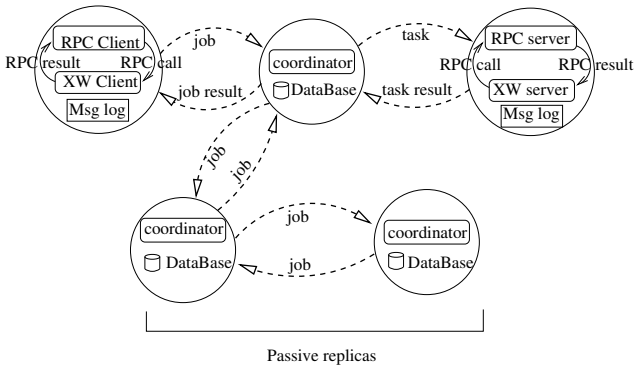

Figure 3. RPC-V Implementation within XtremWeb

result collection may be executed concurrently, typically with a multi-threaded client, in order to start collecting results while submission is still on going.

The application, programmed using a subset of the GridRPC API, runs concurrently with the XtremWeb client and issues RPC calls to it.

As previously discussed, the XtremWeb client transparently logs every message and tags them with a timestamp. Several logging strategies can be considered. The performance section will compare optimistic, blocking pessimistic and non-blocking pessimistic logging. The synchronization mechanism is implemented in the core of the XtremWeb client.

Coordinator. The Coordinator schedules the client RPC calls to the appropriate server, according to a set of criteria: service availability, server workload, observed parameters/results communication time.

The current implementation of the coordinator considers a basic first-come first-serve scheduling policy and a simple coordination scheme between coordinator schedulers: to prevent too much tasks duplication when several server partitions are connected to different coordinators, tasks are replicated among coordinators with their state (finished, ongoing, pending) and each local scheduler takes the following decisions: finished tests are not scheduled by a coordinator replica; ongoing tasks are not scheduled until the coordinator replica suspects the disconnection of its predecessor; pending tasks are scheduled. However, even with such policy, the system asynchrony may lead to duplicated executions. This simple implementation does not schedule RPC redundantly in order to anticipate potential failures. However, this could be added easily with a replication flag associated with the task state.

The current implementation of passive replication connects the replica on a ring topology. Each coordinator knows a set of other coordinators through its neighbors list. Using a common order on this set, a coordinator computes its position in this list, and a successor relationship. Regularly (with the "heart beat" signal), a coordinator sends an abstract of its state to the successor in the list. If the successor does not acknowledge the state propagation, it is suspected to be down, the list is locally updated accordingly and the next coordinator in the list is contacted. Thus, the ring topology is virtual and may change at each "heart beat" signal sent.

For message logging inside the coordinator, we distinguish between job descriptions and file archives. Job descriptions are stored in a database, for fast management, and file archives are stored in an optimized file system. Job descriptions are translated in tasks descriptions stored in the same database, and there is no replication of file archives.

Server. The management of off-line computing servers directly derives from the connection-less communication protocol between the servers and the coordinator. The same server may disconnect the coordinator, continue the execution and re-connect the coordinator later for sending RPC results.

The server receives the task description along with the command line and file archive and launches the execution of the corresponding executable. When the execution terminates, the server builds an archive of new or modified files (including application outputs) and sends it to the coordinator.

The file archives built as the results of the executions represents the server logs. Thus the logging protocol is necessarily pessimistic. The synchronization mechanism is implemented in the core of the XtremWeb server.

5 Performance Evaluation

In the performance evaluation, we made two types of experimentations. This first set presents the performance of our system in a confined environment where we have the control of all the platform parameters. In a second step, we tested RPC-V in a real life environment : deployment on the Internet and a production application.

5.1 Local Experimentation

In this section, we evaluate the performance of three basic fault tolerance mechanisms (message logging, coordinator replication and synchronization) and the system fault tolerance. The objective is to evaluate the general trends governing system performance and fault tolerance. Performance optimizations and system scalability will be investigated in future works.

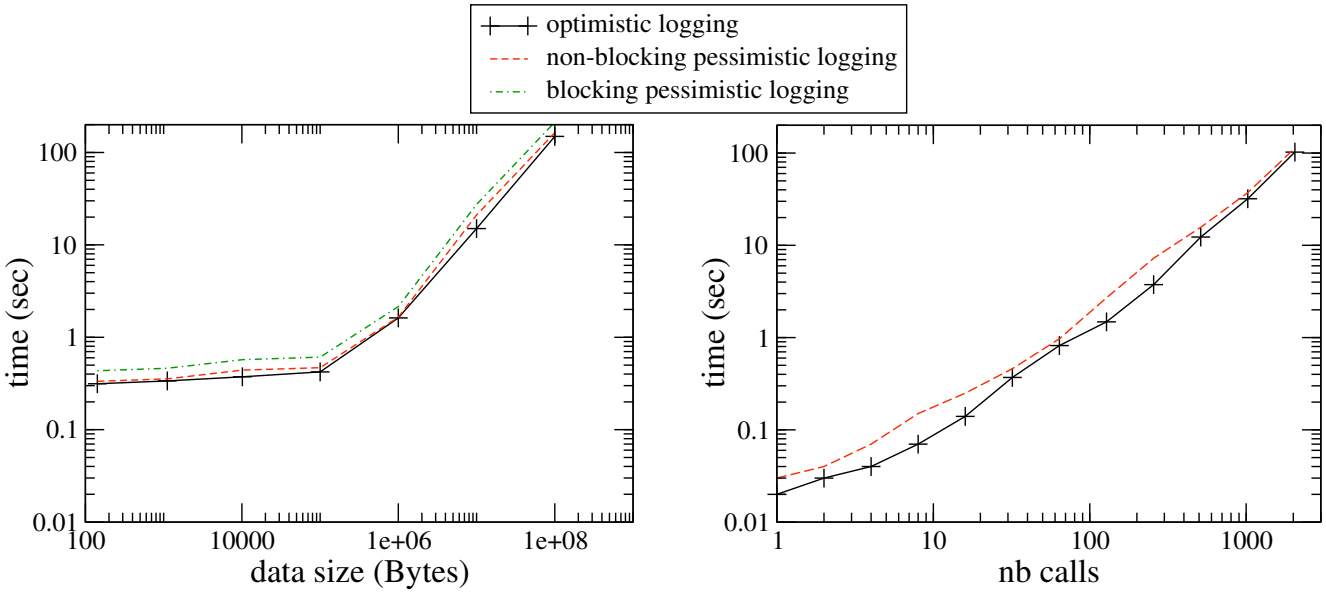

Figure 4. Message Logging

We have selected the experimental platform according to two criteria: 1) experimental results reproducibility and 2) the capacity to highlight the system overheads and fault tolerance capabilities.

A major issue concerning experiments on the Internet is the

experimental conditions and results reproducibility on a network where performance fluctuate widely. We consider that a more controllable environment is preferable.

Highlighting the system overhead implies diminishing the relative impact of all the other components of the execution time. High communication time (communications on the Internet) would hide the system overheads. Thus a more efficient network is desirable.

In order to stress our system to its limits, we have to artificially generate potentially correlated faults at high frequencies which is, on the Internet: 1) difficult to achieve and 2) unlikely to occur during the experiment. For this study too, we need a more controllable network.

All these criteria suggest running the experiments on a fully controllable and isolated platform such as a dedicated cluster, where we can stress our system to a higher degree than on the Internet.

The experimental platform consists in a cluster of PCs under Linux 2.4.20, arranged in three major parts: 16 servers or computing nodes, 4 coordinators and 1 client. All parts are connected to a single 48-port Ethernet 100 Mbit/s switch. All nodes are equipped with Athlon XP 1800+ processors, running at 1.5 GHz, 1 GByte of main memory and an IDE disk.

All experiments run a synthetic benchmark on the client side, executing a set of non-blocking configurable RPC calls. The configuration parameters are the RPC execution time, its parameter and its result size. The RPC execution time will be kept very low (few seconds) in order to highlight the system overheads. The results presented in the figures are mean values over several executions (negligible standard deviations were observed).

To generate faults in a controllable and reproducible manner, we have built a fault generator, running as a remotely controllable daemon. Upon order, or from its own initiative with respect to its configuration, the fault generator kills abruptly the RPC-V component of the hosting machine.

To limit the interference of a too long "heart beat" signal on performance and fault tolerance evaluation, we have set its period to 5 seconds. A fault is suspected when no "heart beat" is received for a period of 30 seconds.

Message Logging: The first experiment compares three different message logging strategies [17] at the client side. The first strategy is the optimistic message logging: logging is done asynchronously, in parallel with the communication. It is optimistic because a crash may occurs before the completion of logging operation. The two other strategies are based on pessimistic logging, either blocking or non-blocking. The blocking one blocks the beginning of the communication until logging completion. The non-blocking one blocks the end of communication until the completion of the logging operation.

For these three protocols, we compare the RPC submission time as measured by the client and we evaluate the synchronization cost that the client will observe. The left part of figure 4 presents the RPC submission time according to the parameters size. We have submitted 16 calls, varying the parameters size from some bytes up to 100 MBytes. The right part of this figure presents the RPC submission time according to the number of RPC calls. Since the size of the

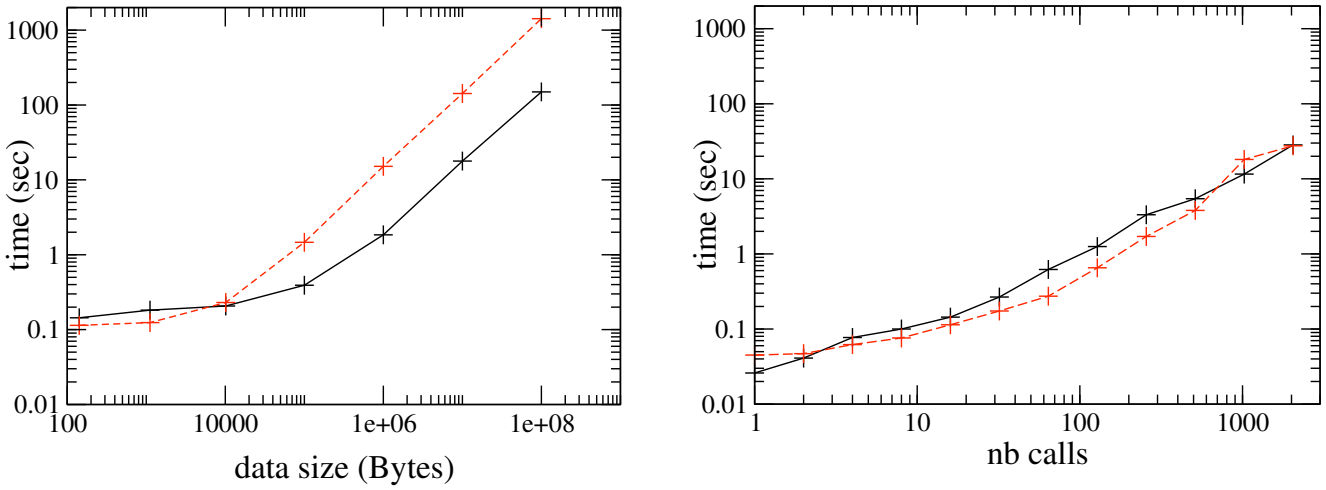

Figure 5. Coordinator Replication Time. Confined: Solid, Real Life: Dashed

RPC calls for this figure are small enough to be marshalled, there is no distinction between the two pessimistic logging protocols.

As expected, the pessimistic blocking protocol adds an overhead of approximately 30%, due to disc accesses. Because optimistic logging protocol runs with a low priority, it adds negligible overhead compared to execution without logging. Because non-blocking pessimistic logging needs a minimal synchronization, it adds small and variable overhead due to disc cache management.

The same fluctuation can be observed on the right figure. In this case, the overhead can reach 100% of the optimistic execution time, because the message logging time equals the communication time for small messages.

The synchronization time depends on the crash pattern of the client and the coordinator. Obviously, there is no synchronization when none of them have crashed. If only one of the components has crashed, the synchronization times for the three protocols are identical: for client restart, the client must re-execute the full application in the three cases; for coordinator restart, there is no differences between the three logging protocols, since client logs can be lost on crash only. When both have crashed, all logs have been lost in the optimistic protocol. Thus, the application has to re-execute all the RPC submissions and the intermediate computations. This is not the case for pessimistic logging where logs can be sent immediately to the coordinator. So the difference is the inter-RPC application computation time.

Coordinator Replication: Solid curves of figure 5 show the coordinator replication time in the confined environment. Every coordinator, has two neighbors: each being a backup for its predecessor. We have measured the time to replicate a coordinator status to its backup according to two parameters: 1) the RPC data sizes with a fixed number of 16 RPCs (leftmost figure) and 2) the number of RPCs to replicate using small size RPCs (∼300 bytes) (rightmost figure).

The leftmost figure shows the impact of database access and the system overhead, which become negligible when the RPC parameters size exceed 1MByte. Up to 10Kbyte, these overheads dominates the replication time.

The rightmost figure demonstrates that replication evolves linearly with the number of task descriptions to replicate. In the current implementation, tasks are replicated one after the other. The optimization consisting of sending the full set of task descriptions would not improve the replication time, on this platform, because it is bounded by database operation time at the backup side.

Synchronization: Figures 6 presents the synchronization time between a client and a coordinator, as measured by the client, when logs are located at the client side or at the coordinator side. Rebuilding the state of the coordinator from the client logs can be six times faster than the opposite. This is due to the implementation of the synchronization, which is asymmetric depending on the location of the logs. When logs are located on the client side, the synchronization operation retrieves the logs list from a local disc access and sends logs to the coordinator. In the second case, the client has to retrieve the logs list from the coordinator, experiencing an additional overhead, before the actual logs exchange begins. The impact of this overhead diminishes when the number of logs or the size of the parameters increases.

Fault Tolerance: To measure our system ability to tolerate faults, we test it under the following conditions: 1 client submits 96 RPCs to a set of four coordinators (actually, only to the preferred one). Each RPC spends 10 seconds and produces few output bytes. 16 servers take in charge the execution of client RPCs. Ideally, total execution would last 60 seconds (6 rounds of 16 parallel RPCs). Depending on activating logging techniques and replication or not, the

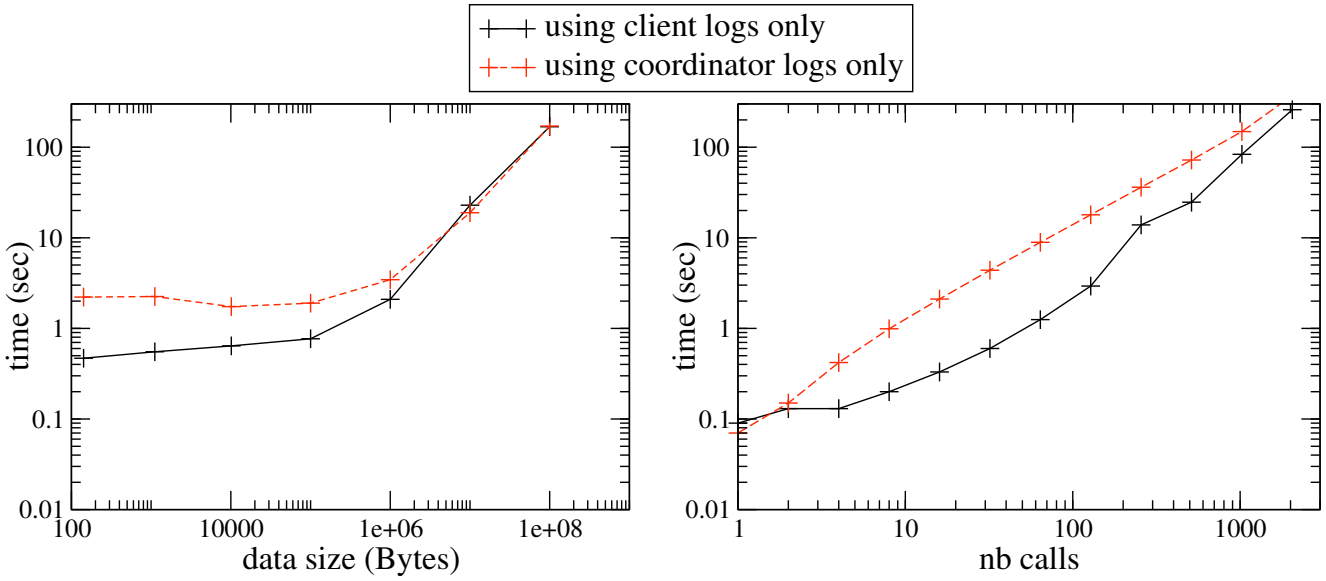

Figure 6. Synchronization Time

overhead of the infrastructure compared to the ideal execution varies between 9 and 11 seconds respectively (about 17% of the execution time) (figure 7). This overhead includes scheduling and data transfers cost. For this test, all nodes of the same kind are running a fault generator, simulating a varying mean time between failures. We considered that faults occurs independently across the nodes. A consequence of this fault generation is the increase of the number of faults in a system for a given time with the number of nodes subject to failure. A fault on server induces the loss of running tasks while a coordinator fault induces the resynchronization of client and all servers.

In figure 7, we compare the impact of the server and coordinator fault on the benchmark execution time. When faults occur, even through the task of restarting coordinator may appear more complex than the one of restarting servers, the dominating parameter is the continuation of the execution at the server side. The two curves clearly demonstrate that the performance degradation is significant in both case, and the server failures have a greater impact than the coordinators failures. The main reason behind this result is the total number of faults which is higher for the servers than for the coordinators. This situation is likely to occur in a real platform, where computing components will dominate the infrastructure ones.

Note that the duration of the server task determines the lowest fault period on servers under which the application does not progress anymore. For the coordinators, this threshold is defined by the minimal duration to ensure the progress condition. Two implications of this result are the following: 1) the full system provides better performance when the most stable nodes are reserved for the servers and not for the coordinators, 2) since the coordinator is in the critical path between the client and the server, and since its availability and performances directly determine the capacity of the application to progress, its basic forwarding functionality should be prioritized compared other mechanisms.

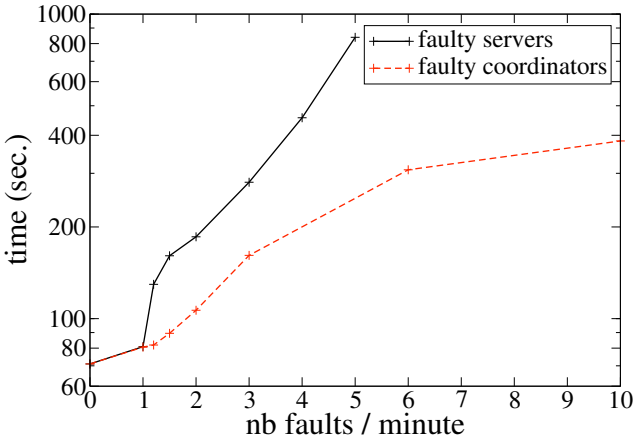

Figure 7. Benchmark Execution Time According to Fault Frequency

5.2 Real life Experiments

To validate the results of the previous experiments, we deployed RPC-V on the Internet. Three Universities participated to this experiment: University Polytechnic School of Lille (France), The University of Wisconsin (USA) and Paris Sud University (France). We installed in each of these places a hundred RPC-V servers in Desktop Linux PCs (about 120) or computing nodes (around 160).

Two dedicated servers are used as coordinators: the main one hosted at Paris is equipped with an Intel Xeon 2.40GHz processor, 1Ghz of RAM and 200GB of Hard disk; the sec-

ond, located at Lille, has approximatively the same configuration (Intel Xeon 2.66Ghz, 1Ghz RAM and 100GB HD). All these computing equipments communicate between each other through the Internet.

For the experiment, we used a real life production application of Alcatel. This application is a tool helping to validate and evaluate commutation networks. It computes the signal lost and the bandwidth for network configurations.

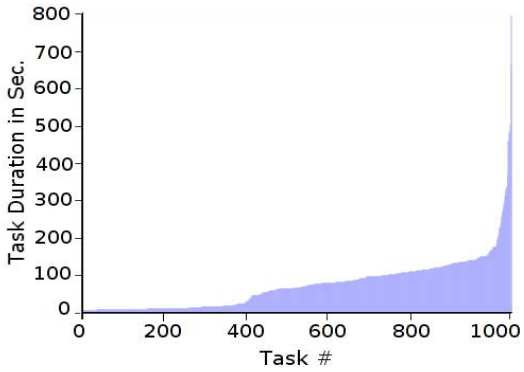

Figure 8. Distribution of Tasks Durations in the Alcatel Application

This application allows the user to set the number of parallel tasks for a given execution. We run this application with 1000 tasks. Figure 8 presents the distribution of the tasks durations for this application and the used set of parameters. Note that the tasks duration varies in a wide range. For all the following tests, the coordinator replication period is set to 60 seconds.

Replication time: The first experiment analyzes the cost (in time) of the replication through the Internet according to the tasks size. We compare data transfers across Internet to data transfer in the confined environment. The dashed curves of the figure 5 (leftmost side), demonstrates that Internet data transfer evolves linearly but the reduced bandwidth limits the replication performance compared to the confined environment.

The second experiment measures the replication cost through the Internet according to the number of tasks description to replicate. Like for replication in the confined environment, the database accesses dominate the replication time. Because the coordinators used for the real life experiments exhibit better performance on database operations, the replication time is lower than the ones in the confined environment.

Fault tolerance: For the three following tests, we run the Alcatel application on the testbed using two replicated coordinators: one at LRI (Paris Sud University - Orsay) and the other one at Lille (about 300 Km between them). The passive replication duplicates asynchronously the Lille coordinator state on its replica at LRI. A single client submits 1000 tasks to the Lille coordinator. We plot the number of completed tasks as seen by the coordinator, according to the time in seconds.

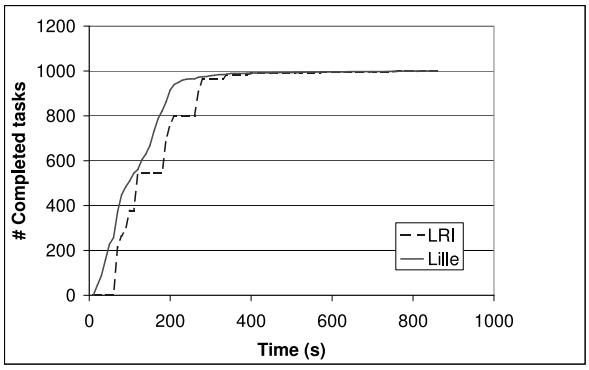

Figure 9. Reference Execution without Fault

In the first test without fault, all servers get their jobs and send their results at Lille (figure 9). We will consider this execution as the reference. The discrete nature of the replication, triggered every 60 seconds is illustrated by the plateaux on the LRI curve.

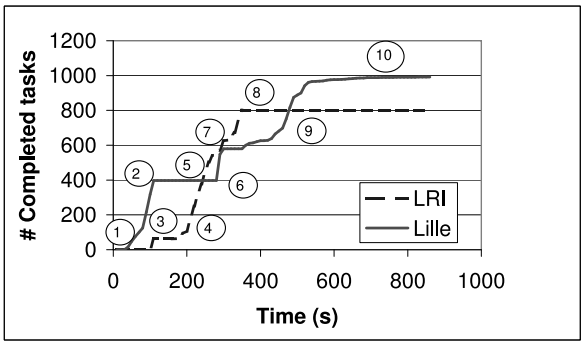

Figure 10. Execution with Two Consecutive Coordinator Faults

The second test measure the perturbation occasioned by the coordinator failure. Figure 10 presents the number of terminated tasks for all minutes of the execution. We start the coordinators simultaneously (label 1). We stop artificially the Lille coordinator when about 400 tasks are completed (label 2). Replication of LRI coordinator starts 60 seconds after the beginning (label 3) but during the replication, the Lille coordinator failed. The plateau is due to the delay experienced by the servers to suspect the failure of the Lille coordinator. The LRI coordinator starts receiving

results from the servers (label 4). The number of tasks received at LRI reach the one received at Lille before the failure (label 5). When all servers have changed their preferred coordinator, we restart Lille (label 6). At that moment, the client and all servers still consider LRI as their prefered coordinator. The Lille coordinator reach a state close to the LRI one (label 7) and then stop replication waiting for the inter replication delay (60 seconds). Then we stop LRI (label 8). The client and servers suspect the fault of the coordinator and contact Lille (label 9). The test terminates using the Lille coordinator (label 10). This figure demonstrates that the system tolerates multiple coordinator faults.

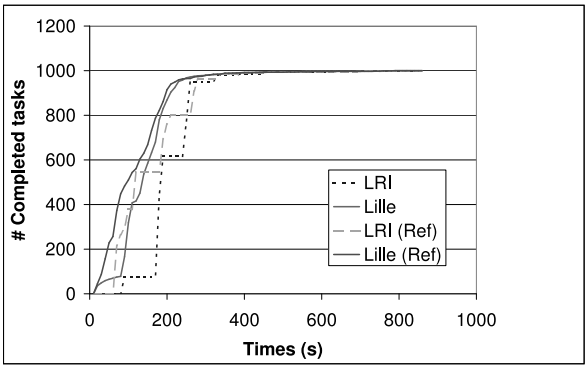

Figure 11. Execution Under a Suspected Partitioned Environment

In a second test we simulate an inconsistent view of the system by its different components: the servers suspect Lille coordinator as faulty, the client suspects LRI coordinator as faulty and the two coordinators considers the other one as running. To implement this test, we hide the existence of the Lille coordinator to the servers and we force the client to submit its RPC calls to Lille. The LRI coordinator still works as a replica of the Lille one, enabling the tasks and results to flow from the client to the servers. Figure 11 compares the number of terminated tasks for all minutes of the execution in this situation with the reference execution. This last test demonstrates that RPC-V can cope with system partitioning, where the components have a different view of the system, as long as there is a path between the client and the servers.

6 Conclusion and Future Work

We have investigated the issue of fault-tolerant RPC on large scale Grids connected by best effort networks such as the Internet. Our first analysis concerned the characterization of these environments as distributed systems. Considering their main features, we conservatively decided to classify them as asynchronous systems subject to intermittent and permanent failures. We defined a novel automatic and transparent fault-tolerant RPC protocol for stateless services, based on a three-tier architecture, unreliable fault detectors, passive replication and message logging. We have discussed the implementation of this protocol within the XtremWeb Desktop Grid middleware.

The evaluation on confined environment and real life conditions has demonstrated the low overhead and fault tolerance of the resulting implementation. The most significant result is the ability of any component of the system to join or quit it without affecting the correct behavior of the rest of the system. The client application progresses as long as there is a path between a client and a server, which may involve several coordinators. A second learning is the higher impact of server faults compared to coordinator (middle tier) faults on the client application execution time. This result is linked to the RPC execution time on the server side, the time to cross the critical path between the clients and the servers (potentially involving coordinator replication) and the mean time between failures of the system components. As long as the RPC execution is longer than crossing the critical path between clients and servers, it is preferable to reserve the most stable resource for the servers. A third result concerns the logging technique on the client side. We demonstrated that non blocking pessimistic logging does not increase the submission time significantly compared to optimistic logging while potentially allowing a shorter re-submission time when client and coordinator have crashed.

As perspectives, we plan to evaluate the system optimizations and scalability on a larger Desktop Grid testbed harnessing thousands of nodes. We plan to study the impact of checkpointing server tasks on performance and fault tolerance. We will also investigate fault tolerance RPC in other contexts such as classical RPC and Grid Services.

References

[1] http://www.edonkey2000.com.

[2] P. Arbenz, W. Gander, and M. Oettli. The Remote Computation System. In *Parallel Computing*, volume 23, pages 1421–1428, 1997.

[3] Omar Bakr and Idit Keidar. Evaluating the running time of a communication round over the internet. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 243–252. ACM Press, 2002.

[4] R. Baldoni, C.Marchetti, and S.Tucci Piergiovanni. Asynchronous Active Replication in Three-tier Distributed Systems. In IEEE Computer Society, editor, *Pacific Rim International Symposium on Dependable Computing (PRDC'02)*, page 19, Tsukuba, Japan, December 2002.

[5] Roberto Baldoni and Carlo Marchetti. Three-tier replication for FT-CORBA infrastructures. In *Software - Practice and Experience*, pages :33:767–797, 2003.

[6] Ziv Bar-Joseph and Michael Ben-Or. A Tight Lower Bound for Randomized Synchronous Consensus. In *Annual ACM Symposium on Principles of Distributed Computing*, pages 193 – 199, 1998.

[7] Micah Beck, Jack Dongarra, Jian Huang, Terry Moore, and James S. Plank. Active logistical state management in gridsolve. In IEEE Computer Society, editor, *4th International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, 2003.

[8] H. Casanova and J. Dongarra. NetSolve's Network Enabled Server: Examples and Applications. In *IEEE Computational Science and Engineering*, 5(3):57-67. September 1998.

[9] Henri Casanova and Jack Dongarra. NetSolve: A Network-Enabled Server for Solving Computational Science Problems. In Sage Publications, editor, *The International Journal of Supercomputer Applications and High Performance Computing*, volume 11, Number 3, pages 212–223, 1997.

[10] Henri Casanova, MyungHo Kim, James S. Plank, and Jack J. Dongarra. Adaptive scheduling for task farming with grid middleware. *The International Journal of High Performance Computing Applications*, 13(3):231–240, Fall 1999.

[11] Henri Casanova, Graziano Obertelli, Francine Berman, and Rich Wolski. The AppLeS parameter sweep template: User-level middleware for the grid. In *ACM/IEEE International Conference on SuperComputing SC 2000*, pages 75–76, 2000.

[12] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end wan service availability. In *Proc. 3rd USENIX Symposium on Internet Technologies and Systems (USITS*, pages 97–108, San Francisco, CA, 2001.

[13] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[14] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46, 2001.

[15] J. Czyzyk, M. Mesnier, , and J. More. The NEOS Server. In IEEE Computer Society, editor, *IEEE Journal on Computational Science and Engineering*, volume 5, Number 3, pages 68–75, 1998.

[16] S. Djilali. P2p-rpc: Programming scientific applications on peer to peer systems with remote procedure call. In IEEE Press, editor, *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*. Tokyo Japan, November 2003.

[17] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[18] G. Fedak, C. Germain, V. Neri, and F. Cappello. XtremWeb: A Generic Global Computing System. In IEEE Computer Society, editor, *IEEE Int. Symp. on Cluster Computing and the Grid*, pages 582–587, Brisbane, Australia. 2001.

[19] C. Fetzer and S. Mishra. Transparent TCP/IP based Replication. In *Proceedings of the 29th International Symposium on FaultTolerant Computing*, Madison, Wisconsin, June 1999.

[20] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. In *Journal of the ACM*, pages 32(2):374–382, April 1985.

[21] Indranil Gupta and Ken Birman. Building scalable solutions to distributed computing problems using probabilistic components. In *The International Conference on Dependable Systems and Networks (DSN-2004), Dependable Computing and Computing Symposium (DCCS)*, June 28 - July 1, 2004. Florence Italy.

[22] G. Tsudik S. Tuecke I. Foster, C. Kesselman. A Security Architecture for Computational Grids. In *5th ACM Conference on Computer and Communications Security Conference*, page 1998, 83-92.

[23] H. Nakada, M. Sato, , and S. Sekiguchi. Design and Implementation of Ninf: toward a Global Computing Infrastructure. In *Future Generation Computing Systems, Metacomputing Issue*, volume 15, pages 649–658, 1999.

[24] A. Natrajan, M. Humphrey, and A. Grimshaw. Grids: Harnessing Geographically-Separated Resources in a Multi-Organisational Context. In *Proceedings of the 15th Annual Symposium on High Performance Computing Systems and Applications(HPCS 2001)*, Ontario, Canada, June 2001.

[25] James S. Plank, Henri Casanova, Micah Beck, and Jack J. Dongarra. Deploying Fault Tolerance and Task Migration with NetSolve. In Sage Publications, editor, *Future Generation Computer Systems*, pages 15:745–755, 1999.

[26] M. O. Rabin. Randomized byzantine generals. In *Proc. of the 24th Annu. IEEE Symp. on Foundations of Computer Science*, pages 403–409, 1983.

[27] R. Raman, M. Livny, and M. Mutka. Condor: a Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Systems*, June 1998.

[28] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[29] Mitsuhisa Sato, Motonari Hirano, Yoshio Tanaka, and Satoshi Sekiguchi. OmniRPC: A Grid RPC facility for cluster and global computing in OpenMP. *Lecture Notes in Computer Science*, 2104:130–136, 2001.

[30] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In Springer (2002), editor, *Proceedings of the Third International Workshop on Grid Computing (GRID 2002)*, pages 274–278, Baltimore,MD, USA, 2002.

[31] Paul Stelling, Cheryl DeMatteis, Ian T. Foster, Carl Kesselman, Craig A. Lee, and Gregor von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128, 1999.

[32] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory tcp: Highly available internet services using connection migration. In *Proc.of ICDCS*, volume 4, pages 17–26, 2002.

[33] Paul Townend and Jie Xu. Fault tolerance within a grid environment. In *Proceedings of AHM2003, http://www.nesc.ac.uk/events/ahm2003/AHMCD/pdf/063.pdf*, page 272, 2003.

[34] Werner Vogels, Robbert van Renesse, and Ken Birman. The power of epidemics: Robust communication for large-scale distributed systems. In *HotNets-I '02: First Workshop on Hot Topics in Networks, special issue of the ACM SIGCOMM Computer Communication Review*, Princeton, NJ. October 2002.