Fault Tolerance Management for a Hierarchical GridRPC Middleware

Aurelien Bouteiller, Frederic Desprez
LIP ENS Lyon
UMR 5668 CNRS-ENS Lyon-INRIA-UCBL, F-69364 Lyon Cedex 07
aurelien.bouteiller@inria.fr

***Abstract*—The GridRPC model is well suited for high performance computing on grids thanks to efficiently solving most of the issues raised by geographically and administratively split resources. Because of large scale, long range networks and heterogeneity, Grids are extremely prone to failures. GridRPC middleware are usually managing failures by using 1) TCP or other link network layer provided failure detector, 2) automatic checkpoints of sequential jobs and 3) a centralized stable agent to perform scheduling. Most recent developments have provided some new mechanisms like the optimal Chandra & Toueg & Aguillera failure detector, most numerical libraries now providing their own optimized checkpoint routine and distributed scheduling GridRPC architectures. In this paper we aim at adapting to these novelties by providing the first implementation and evaluation in a grid system of the optimal fault detector, a novel and simple checkpoint API allowing to manage both service provided checkpoint and automatic checkpoint (even for parallel services) and a scheduling hierarchy recovery algorithm tolerating several simultaneous failures. All those mechanisms are implemented and evaluated on a real grid in the DIET middleware.**

***Index Terms*—GridRPC, Fault tolerant, Failure detector, Checkpoint, Distributed algorithm.**

I. Introduction

Because grids are gathering a wide variety of computing, storage, and network resources, coming from several geographically distributed sites, it is especially challenging to use those platforms for high performance computing applications. Among existing computing models over a grid, one simple, powerful, and flexible approach consists in using servers available in different administrative domains through the classical client-server or Remote Procedure Call (RPC) paradigm. Network Enabled Servers (NES) [1], [2], [3] is a family of middleware implementing the GridRPC [4] API. Clients submit computation requests to a scheduler whose goal is to find a server available running a given computation service over the grid. Scheduling is frequently applied to balance the work among servers and a list of available servers is sent back to the client; the clients are then able to send the data and the request to one of the suggested servers to solve their problem.

Another challenging issue in grids is reliability: when the number of components of an architecture increases, the mean time between failures (MTBF) decreases accordingly; grids are by nature gathering more resources than clusters. Heterogeneous components of a grid are even more prone to failure because of mixed flavors of hardware or slight differences of implementations in interoperating software from different suppliers or operating systems. Moreover grids are using long range networks where packet loss are common. Intermediate routing peers may also introduce unexpected slowdown on message speed. As a consequence, failures are not uncommon events anymore and production deployments have been facing unreliability issues [5]. This strengthen the need for a grid convenient management of failures for any NES middleware focusing on large scale platforms. The usual way to deal with failure in NES systems is to rely on the transport layer (like TCP) to detect failures of peers. Then, the corrective action is whether to reschedule the lost tasks; whether, for the most advanced ones, to restart from checkpoint to decrease the amount of lost computation. Because the grid infrastructure is usually centralized, nothing is done to cope with failures of the scheduler.

All of those three aspects needs to be improved to address the challenges raised by modern grids. 1) In grids, relying on TCP heartbeats leads to long failure detection time (hours timeouts) and poor accuracy, which in turns leads to low throughput in an unreliable environment. 2) Many grid services are bindings of well-known numerical library: a single call to a routine might trigger a full scale parallel job (ScaLAPACK is an example). Some libraries provide their own optimized checkpoint routine; still NES have to preclude loss of recovery data with the service resource. Middleware proposing checkpoints could only manage sequential jobs so far, raising the need for a simple yet flexible checkpoint interface to manage all of those techniques. 3) Recent developments in GridRPC systems have demonstrated the major performance improvement of using a distributed scheduling architecture instead of a centralized scheduler [6]. The DIET [3] project is the first NES middleware proposing a scalable architecture based on several hierarchies of agents. Recovering this architecture requires a distributed fault tolerant algorithm between the agents.

In this paper we describe and evaluate experimentally in DIET three fault tolerant mechanisms intended to solve those issues. We present the first implementation and evaluation in a grid of the Chandra & Toueg & Aguilera [7] optimal failure detector. Then we design a novel checkpoint interface between the NES and the gridRPC middleware, providing automatic checkpoint to non fault tolerant aware services (even parallel ones) and reliable distributed grid storage of recovery data

to self checkpointing ones. Last we propose and evaluate a distributed recovery algorithm rebuilding the scheduling agent hierarchy when several failures can occur simultaneously.

The rest of this paper is organized as follows. The next section discuss the basics of a gridRPC middleware by depicting the architecture of DIET as an example. Then related works section outlines the originality of our proposed mechanisms. The third section presents the novel checkpoint API and how it can manage automatic checkpoint of parallel services. Then the next section defines the distributed algorithm for scheduling hierarchy recovery. Sixth section gives an overview of the failure detector algorithm used in DIET. Seventh section presents experimental evaluation of those mechanisms outlining their efficiency in a real grid deployment. Last we conclude and discuss future works.

II. The GridRPC Context: The DIET Example

The aim of a GridRPC middleware is to provide a toolbox that will allow different applications to be ported efficiently over the Grid and ease access to distributed and heterogeneous resources. Several middleware have been developed to fulfill those requirements; the architecture of every NES system relies on three main entities: the servers offering computational services to the grid, the clients using the grid to solve their problems, and the infrastructure nodes matching the client needs and the services offered by computing resources. DIET is a good example of a production quality NES software as it shares this basic architecture but also includes state of the art distributed scheduling architecture. In this section we describe the DIET architecture to better understand the fault tolerant requirement induced by every GridRPC middleware.

A **Client** is an application that uses DIET to solve problems using an RPC approach. Users can access DIET via different kinds of client interfaces: web portals, PSEs such as Scilab, or from programs written in C or C++. A **SeD**, or server daemon, provides the interface to computational servers and can offer any number of application specific computational services. A SeD can serve as the interface and execution mechanism for a stand-alone interactive machine, or it can serve as the interface to a parallel supercomputer by providing submission services to a batch scheduler. All the DIET entities uses Corba to communicate.

Agents provide higher-level services such as scheduling and data management. These services are made scalable by distributing them across a hierarchy of agents composed of a single **Master Agent (MA)** and several **Local Agents (LA)** as shown in Figure 1. In order to access DIET scheduling services, clients only need a string-based name for the MA (e.g. "MA1") they wish to access; this MA name is matched with a Corba identifier object via a standard Corba naming service. Clients submit requests for a specific computational service to the MA. The MA then forwards the request in the DIET hierarchy and the child agents, if any exist, forward the request downward until the request reaches the SeDs. The SeDs then evaluate their own capacity to perform the requested service; capacity can be measured in a variety of

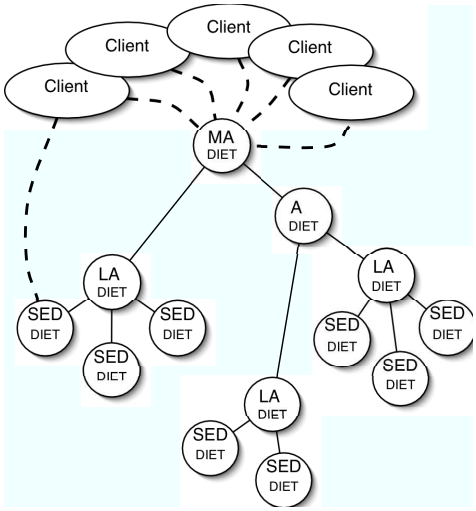


Fig. 1. DIET hierarchical organization.

ways including an application-specific performance prediction, general server load, or local availability of data-sets specifically needed by the application. The SeDs forward their responses back up the agent hierarchy. The agents perform a distributed collation and reduction of server responses until finally the MA returns to the client a list of possible server choices sorted using an objective function (computation cost, communication cost, machine load, ...). The client program may then submit the request directly to any of the proposed servers, though typically the first server will be preferred as it is predicted to be the most appropriate server.

This architecture emphasis why we need to focus on two aspects. Clients can be restarted from an external process and results hold back until the restarted client collect them. Conversely without architecture recovery, resources disconnected from the MA are never used by the scheduler and the platform throughput reduced. Without SeD recovery, large amount of time is lost recomputing several time the same service. Because those two procedures are triggered by detection of failed processes, a fast failure detector is a requirement to any efficient recovery.

III. Related Works

A first approach to service recovery in RPC-like systems is simple resubmission of lost jobs. Unfortunately this leads to lose lot of elapsed computation time. Some global computing platforms like SETI and BOINC [8] use process replication to recover from computing resources crash. In replication, the same job is running on several hosts and an atomic broadcast based protocol ensures consistency of replicas. When a failure hits some processes, the recovery procedure is to rebuild new replicas. Because of the voluntary social model of the global computing platform, overall available computing power far oversize the application needs and replication comes at no cost. However resources cannot be trusted; therefore the inherent detection of process corruption provided by replication is very appealing. Back in a classical grid system, compared to checkpoint, replication would divide the available computing

power by the replication factor and induce a several order of magnitude performance decrease in realistic deployments.

Checkpoint based approach has already been used in Ninf [9]. The Condor Standalone Checkpoint Library is used to build sequential process snapshot. Checkpoints are stored on a stable centralized checkpoint server holding all recovery data. Compared to our distributed approach, this checkpoint server is a potential performance bottleneck and a single point of failure. NETSOLVE [10] address this issue by replicating checkpoint data in computing nodes volatile memory. Our distributed mechanism spares the scarce and performance crucial memory by using disk space to hold checkpoint data. To our knowledge our checkpoint approach is the only one able to manage service provided checkpoint and automatic checkpoint of parallel services.

DIET is the only NES with distributed architecture. Thus we provide the only implementation and evaluation of a distributed recovery algorithm for hierarchical scheduling.

With failure detectors used in other gridRPC systems, even with hours timeouts, production deployments still experience large amount of false detections. The failure detector implemented in DIET detects failures faster and can adapt the network overhead it produces on the fly accordingly to detected message loss probability and average delay. Because this is the first implementation of this detector in a NES system we are providing the first performance evaluation in a real grid.

IV. BACKING UP SERVICE PROGRESS

Failures hitting computing resources have a larger impact on application performance compared to those affecting infrastructure. This is mainly due to the large amount of lost computation time when using large grain RPC [11]. Checkpoint is an efficient mechanism to avoid restarting crashed jobs from the beginning. An efficient grid checkpoint mechanism has to be able to manage, automatic, service provided, and parallel checkpoints. Recovery data must be preserved in some grid-wise persistent storage to preclude their disappearance with failed services.

A. Automatic checkpoint of sequential jobs

There is various checkpoint libraries creating a full disk image of a process, like the Condor Standalone Checkpoint Library (CSCL) [12] or the Berkeley Lab Checkpoint/Restart (BLCR) [13]. They dump to a file the complete process memory, stack and processor registers; this image can be restarted on any similar architecture. Service code does not need to be modified to integrate fault tolerance, it only has to be linked with the checkpoint library. Therefore it is efficient at reducing software development cost, which is an important issue in grid exploitation. In DIET we currently use CSCL but plan to migrate to BLCR soon.

B. Storing recovery data in a persistent storage

Recovery data such as checkpoints are useful only if they are still available after failure. This can be achieved by 1) using a centralized, stable, checkpoint server, holding all the recovery data or 2) distributing several replicas of the recovery data on the computing nodes themselves. We use the second approach as it is more scalable and does not have a single point of failure.

JUXMEM [14] is a grid data sharing software addressing the issue of persistence across failures. Data are replicated on several hosts to ensure that some copies still exists when failures occur on the repositories. In DIET, all the SeD created recovery data are transferred on the fly in the JUXMEM distributed data storage.

C. Rescheduling failed jobs

The client is in charge of most service recovery tasks: it monitors the services running the RPC it has initiated and reschedules the failed ones as soon as possible. The client management of failures is embedded in the RPC mapping layer of the grid library and is processed without client code interaction. The fault tolerance is always automatic from the client point of view, independently of the selected checkpoint technique for services.

Technically, in DIET, when a client initiates a new RPC, the gridRPC library first contacts the Master Agent to get some matching computing resource. It submits the RPC and observes the selected SeD using a fault detector. Each time the SeD has completed a new checkpoint, it sends the JUXMEM data identifier to the client yet checkpoint data itself never goes to the client. When the client library detects a failure, it requests a new SeD to the MA and the RPC is restarted from the last checkpoint.

D. Common interface for automatic, parallel, and service provided checkpoint

Common point between all the checkpoint techniques is that they generate files. However, in service provided checkpoint, the grid middleware is not in charge of generating checkpoints. The service itself provides a suitable and more efficient checkpoint routine. Still the checkpoint data needs to be preserved in the persistent storage. The same holds for parallel services. This leads a generic checkpoint service to have a file centric approach; providing a convenient procedure for the service to register any files to be included in the recovery data set, and to notify when those data are ready. In DIET, the checkpoint interface includes four functions:

- `DietCkpt_RegisterFile`: this function allows a service provided checkpoint mechanism to point out any data required to recover from failure. This function may also be used to embed shell scripts suitable for restarting the failed process.
- `DietCkpt_Notify`: this function is called by the service provided checkpoint mechanism when local checkpoint data pointed by the previous function has been successfully updated. All registered files are merged into a service checkpoint and transferred to the persistent data storage.

- `DietCkpt_CkptNow`: This is basically a convenience function build on top of the two previous to provide semi-automatic checkpoint. It lets a service self-checkpoint immediately, using the Condor checkpoint library. The application decides when it is wise to take a checkpoint but do not need to manage how.
- `DietCkpt_Automatic`: this function initializes a timer to trigger periodically the `DietCkpt_CkptKnow`. This provides fully automatic checkpoint.

E. Automatic checkpoint of parallel jobs

A grid service can also be a front-end to a parallel application. The ScaLAPACK library is a good example: a single function actually starts a fully distributed application using the Message Passing Interface (MPI) to manage communications between processes. It is out of the scope of a grid middleware to manage internal processes and communications of the service, thus distributed processes not providing their own routine to build a coherent -recoverable- set of process checkpoints cannot be restarted.

Hopefully, some efforts have been developed in MPI libraries to provide automatic fault tolerance; among them MPICH-V [15] is able to build a coherent checkpoint recovery set from any MPI application. Using the interface proposed in previous paragraph, we designed a SeD capable of interacting with MPICH-V. The SeD builds a configuration file emplacing MPICH-V's checkpoint server on its own host; the MPI job is then started as usual. During execution, the SeD controls the checkpoint pace trough a socket connexion with MPICH-V's checkpoint scheduler. Because checkpoint server is located on the same node, checkpoint files are always locally available to the SeD and are registered for duplication using the `DietCkpt_RegisterFile` function. Thanks to this original cooperation between grid and MPI fault tolerant middleware, any MPI parallel service can be managed by DIET.

V. Agents Topology Recovery

The main task of a gridRPC system is to establish a relationship between clients submitting jobs and computing resources. Distributing the scheduling algorithm on a tree hierarchy allows to reach better performance than using a centralized scheduler [16]. However when some sub-trees are disconnected from the architecture by failures, all computing resources they manage become unavailable to clients and the computing throughput is reduced. A distributed algorithm among agents can help to rebuild a connected architecture.

A. Model

The distributed scheduling architecture is a never-ending service. Therefore we propose to use a model similar to self-stabilization [17]. A program is self-stabilizing if 1) starting from any arbitrary initial state (a failure), it eventually reaches a legitimate global state and 2) from any legitimate global state the program execution leads to reach only a legitimate state. In other words the property "the program is in a legitimate state" is stable and eventually true. Although self-stabilization is achievable as soon as a long enough failure free period occurs, no process in the system can ever *know* whether the system has self-stabilized, and during recovery some properties of the program may be false.

We restrict this model by considering the following stronger hypothesis: the network is pseudo-synchronous and the only type of failures are process crash and definitive link shutdown. Transient network failures such as message loss or message modification are supposed to be addressed at a lower -hardware or software- network level. Non reachable agents due to network failures are considered as failed. These strong hypothesis have two main consequences: failed agents can be detected, but we cannot assume arbitrary initial state.

B. Tree rebuild Algorithm

The basic idea of the algorithm is to reconnect the tree by keeping a list of ancestors on each agent. When an agent detects the death of its father, it tries to reconnect to some ancestor upper in the hierarchy. If too many agents fail *at the same time*, an agent could know only failed ancestors. This vulnerability period is the consequence of restraining the recoverable initial states.

1) **Definition: legitimate global state.** A legitimate global state of the algorithm is a state where all non failed agents are organized on a tree topology and each knows at least f ancestors or an entire path up to the root.
2) **Definition: recoverable initial state.** A recoverable initial state of the algorithm is a state were all non failed agents are organized on a forest topology and knows at least one non-failed agent. An eventually reliable oracle locating the root exists.

Every agent except the root is running the algorithm presented in Figure 2. When detecting the failure of its parent, the agent tries to reconnect to the nearest (in the tree) alive ancestor. If this ancestor has also failed, it then tries with the second nearest until it is able to reconnect the tree. If the agent has tried every known ancestor of its list and all has failed, then it gets the root location by asking an eventually accurate oracle. The oracle process is an external stable process in charge of locating and observing the root of the hierarchy. Each time an agent becomes the new father of a subtree, it broadcasts its own ancestors list downward to update the list of the agents knowing less than f alive ancestors.

This algorithm is always able to recover without central coordination from at most $f-1$ simultaneous failures, where simultaneous means occurring before a full recovery. Actually it can recover from more failures most of the time, as long as a there do not exist a consecutive chain of f failed ancestors. Despite eventually all resources are available to clients, infrastructure provides a best effort service: during recovery some computing resources may not be reached and due to failure, some requests might be totally lost.

```
initialization
  ∀i ∈ [1..f], Φ[i] ← id(ancestor^i(A))
  i ← 1
  observe(Φ[1])
end

# My father died, connect to grandfather
when failed(Φ[1]) then
  i ← i + 1
  if defined(Φ[i]) then
    Φ[1] ← Φ[i]
  else
    # manage connection to root
    Φ[1] ← oracle()
  endif
  connect(Φ[1])
  observe(Φ[1])
end

# New son connected (my child died)
# Updating his ancestors list
when idson ← new_connexion(ANY) then
  add(sonlist, idson)
  send(idson, Φ[1..f − 1])
end

# Received an ancestors list
# Propagate it in my subtree
when recv(Ψ[2..f]) then
  i ← 1
  if ∃ id ∈ Ψ[2..f − 1] ∧ id ∉ Φ[2..f − 1] then
    # Some of my ancestors changed: update my sons
    Φ[2..f] ← Ψ[2..f]
    ∀ idson in sonlist, send(idson, Φ[1..f − 1])
  else
    Φ[f] ← Ψ[f]
  endif
end
```

Fig. 2. The distributed agent tree hierarchy recovery algorithm.

C. Implementation

All DIET components are considered as agents, LA and MA are nodes while SeD are leafs yet also maintaining an ancestors list. Communication between agents relies on RPC and ancestor lists contains Corba identifiers. During initialization a downward broadcast fills the initial ancestor list of agents. Because some request can be lost during architecture recovery, clients reschedule requests based on a timeout.

The oracle is composed of a stable machine hosting both the Corba name service and a small program in charge of observing and restarting the root using the DIET fault detector. As soon as the root gets restarted on a new resource the Corba name service gets locally updated. During failure detection time the oracle might give a wrong answer; this is a normal behavior. As the agent keeps asking until it successfully connects, it will eventually get the correct answer.

Each agent may observe from one to f of its parents depending on a user defined parameter. While knowing an agent is just having its Corba identifier in the ancestor list, observing an agent involves actively monitoring it using a failure detector. An agent does not observe all its known ancestors to decrease network overhead. However there is a tradeoff between the number of observed ancestors and recovery speed: when its grandfather also died, an agent observing only its father would have to wait for two failure detection delays before succeeding in the recovery procedure. By observing both ancestors it can detect failure of both earlier

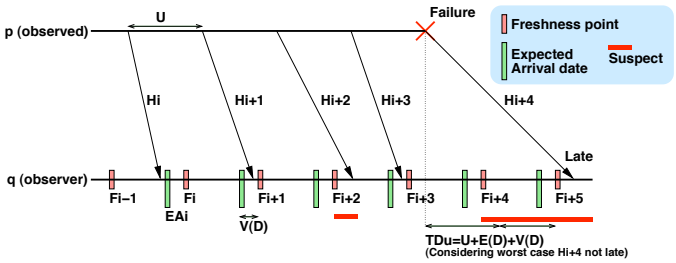


Fig. 3. Sample execution of the DIET fault detector.

(thus stabilize faster). Oracle failure detector is set to detect failed root in at most 1s, while agents are detecting failure of observed ancestors in 5s. Each agent is observing the $\frac{f-1}{4}$ closest ancestors among the f he locally knows. Those default settings may be user tuned through configuration files.

VI. Failure Detection

In the algorithms discussed in the two previous sections, recovery actions are triggered by the detection of some failure. This suggest that time to detect a failure has a major impact on fault recovery efficiency. Pseudo-synchronous message passing is a good model to describe grid systems. Processes communicate trough messages. There is no global clock but there are (unknown) bounds on process speed and maximum message delay. In such systems classical failure detectors suffer from long detection time, poor accuracy and do not even provide good computability [18]. Fortunately unreliable failure detectors of the $\diamond P$ class are more adequate. They have two properties: strong completeness (failed process are eventually suspected by each correct process) and strong accuracy (correct processes are eventually not suspected). In this paper we propose to use the Chandra Toueg & Aguilera detector (CTA-FD) [7]. Quality of service (QoS) of a failure detector can be determined by considering how fast it suspects failures and how well it avoids false detection. Another important parameter is the network overhead from heartbeats. Given a heartbeat frequency and a maximum time to detect a failure, the CTA-FD has an optimal accuracy. Another nice feature is its ability to self reconfigure according to fluctuating network delay and message loss probability.

A. Fault detector's algorithm

Figure 3 presents a sample execution of the fault detector. The basic idea of the algorithm is, given an expected QoS, to compute a heartbeat period U for p and some time intervals $[F_i, F_{i+1})$ for heartbeat H_{i+1} to arrive on q. The QoS parameters are the upper bound on time to detect a failure (TDu), the upper bound on average mistake duration and lower bound on average mistake recurrence time. At time F_i, if q has received a heartbeat $H_j, j \geq i$, then q trusts p for the entire time interval $[F_i, F_{i+1})$. If not, q starts suspecting p until it receives a heartbeat $H_j, j \geq i$.

F_{i+1} is the freshness point of heartbeat H_i: if H_i arrives after F_{i+1} it is discarded. Thanks to freshness points, maximum time to detect a failure does not depend on maximum message delays, but on average message delay E(D); a great

#processes	1	2	3	4	5	6	7	8
Memory (KB)	4	4	4	4	4	4	8	8
Accuracy	1	1	1	1	1	1	1	1
#Processes	9	10	11	12	13	14	15	16
Memory	8	8	8	8	12	12	12	12
Accuracy	1	1	1	1	1	1	1	1
#Processes	32	64	128	256	512	1024		
Memory	24	48	96	184	364	728		
Accuracy	1	1	1	1	.9997	1		

Fig. 4. Failure detector memory footprint and accuracy depending on the number of observed processes (Giga Ethernet).

improvement compared to other failure detectors. Another benefit is independence between heartbeats; a faster than average heartbeat does not trigger a timeout earlier. To set freshness points, three values are computed from the history of received heartbeats: 1) EA_i the expected arrival date of H_i on q, 2) $V(D)$ the variance of message delay and 3) P_l the message loss probability. When network performance are fluctuating, the observing process adapts heartbeat period and freshness points accordingly to history.

B. Implementation in DIET

Fault detector of DIET (DIET FD) is implemented in a standalone C++ library providing 1) a simple interface to observe other services and 2) observable service registration and heartbeat emission. When a new RPC is submitted, the instance of the service registers its process and Corba identifier to the heartbeat library. The observer makes a RPC including the desired heartbeat frequency and the name of the service to start monitoring. As long as the state of the instance is correct (read trough */proc* in Linux), heartbeats are sent.

There is no centralized fault detector in DIET. Agents uses the detector to observe ancestors and clients to observe SeDs. The detector is initialized with pessimistic parameters until a first history is built. Once we received 100 heartbeats, the fault detector reconfigures to match sample properties. Then, every time a heartbeat is received a new period is computed; if it is 10% different from previous one a reconfiguration occurs.

VII. Performance

A. Testbed

The Grid'5000 platform has been used as testbed. It is constituted of 9 clusters distributed all over France and linked trough a 10Gb/s WAN. We used the Paris cluster (260 IBM e325 nodes), the Lyon cluster (64 IBM e325, 64 Sun Fire V20Z) and the Sophia cluster (100 IBM e325). IBM machines have 2 Opteron 2GHz processors, Sun ones have 2 Opteron 2.2GHz. Each node have 2GB DDR memory and giga-Ethernet network. The operating system is Linux Debian using a 2.6.18 kernel. We inject failures by sending UDP orders to a small helper process running on every node; it triggers kill -9 on the target.

B. Failure detector

Because failure detection of SeDs is centralized on the client and a client may submit a large number of simultaneous RPC calls, capability of the failure detector to observe a large number of services is a major issue. Table 4 shows two main parameters of the failure detector considering the number of observed processes: memory footprint and accuracy. Up to 256 processes, client and all observed SeD are located on the Paris cluster. When more processes are observed some are running on Sophia and Lyon. One process is running per node even for dual CPU ones (because our observations are network related). Measurement begins after one hour (to avoid startup effect) and last for one hour.

The first row gives memory footprint of the observer depending on the number of observed processes. The client is instrumented to give the stack and heap memory from the `/proc/pid/statm` file. Memory usage unrelated to fault detection is removed by subtracting memory footprint of the client not observing any process. Memory footprint increases by 4KB steps, due to kernel paging memory allocation algorithm. Aside from this stair effect, the memory footprint increases linearly with the number of observed processes. Average memory used per observed process is 708 bytes as confirmed by the Valgrind tool.

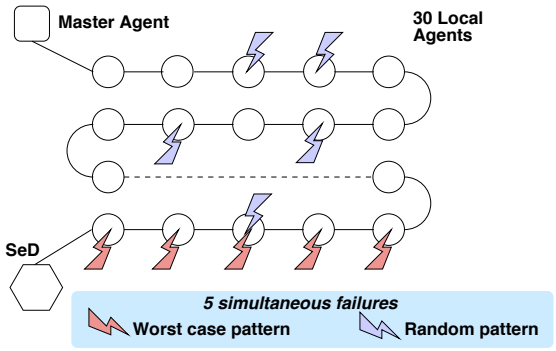
The second row shows the accuracy of the failure detector. Failure detector is set to 1) detect failures in maximum 30 seconds, 2) maximum false detection is 60 seconds, 3) at most 1 false detection occurs per month. In this experiment we do not introduce any failures. When all processes are on the same cluster (up to 256 processes), accuracy is perfect. When using WAN links, accuracy is nearly perfect, whatever is the number of observed processes. Actually we only observed some false detection when observing 512 processes, due to some poor network conditions on long range links during the experiment.

Detected network parameters sets the failure detector to send a heartbeat every 9.8s. Each heartbeat is a 40 bytes message (including UDP headers). 1024 observed processes are sending an overall 33.5Kbit/s of heartbeat data, a very moderate overhead for the 1Gb/s network.

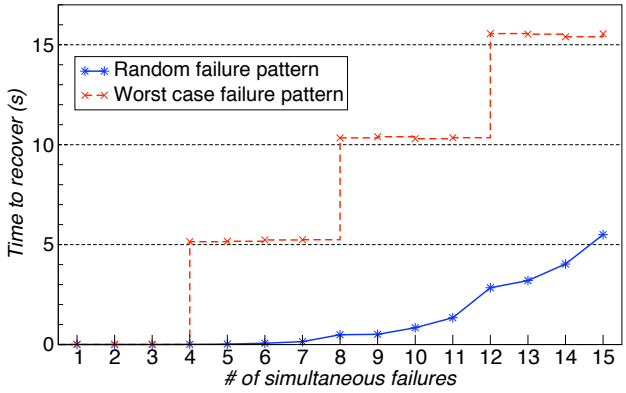
C. Architecture rebuild

Figure 5 shows time to recover after an increasing number of failures hitting the architecture. We use a vertical chain of 32 agents in this experiment (1 MA, 1 SeD, 30 LA) as depicted in Figure 5(a). The number of known ancestors by each agent is set to 16, among those the 4 first are actively observed by the internal DIET failure detector of the agent. Maximum time to detect a failure is set to 5s. Presented values are average of 50 runs measured with the `gettimeofday` system call from the initial detection of the failure to the detection of the full recovery (postmortem by trace analysis).

In the worst case failure pattern we crash ancestors of the lower node in the hierarchy in order, up to reach the desired number of failures. When the number of failure is less than 4 - the number of observed ancestors in this experiment- recovery overhead consists only in transmitting the ancestors lists over the tree. It is the same order of magnitude as RPC calls latency and the overall recovery overhead is clearly dominated by failure detection. When the number of simultaneous failures is greater than the number of observed ancestors, recovery is slowed by the time to detect that the reconnected agent already

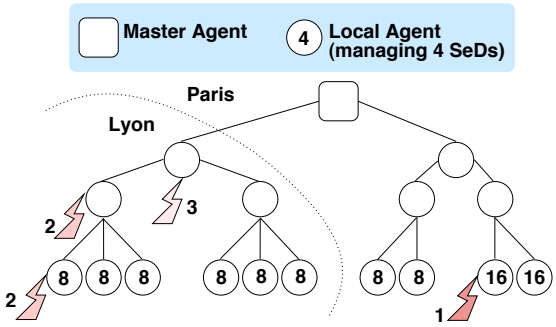


(a) Architecture deployment and failure pattern.

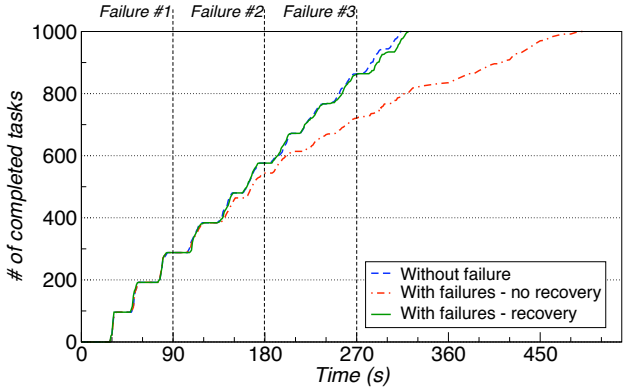


(b) Time to recovery.

Fig. 5. Stabilization time of the agent recovery algorithm when the number of simultaneous failures increases.



(a) Architecture deployment and failure pattern.



(b) Time to complete 1000 jobs.

Fig. 6. Makespan of 1000 BLAS DGEMM depending on the architecture recovery algorithm.

failed. Observing every known ancestors would flatten this overhead at the cost of increasing network load.

Random failure pattern consists in randomly crashing some of the nodes in the architecture. For a small number of simultaneous failures, the probability of running into the worst case scenario is low (less than 1% for 5 simultaneous failures). As a consequence the recovery occurs in parallel for different agents and does not cost additional failure detection time. When the number of simultaneous failures is high, the probability of reaching a worst case scenario increases. It is near 100% to pay one failure detection cycle considering one fourth of the agents are failing. Although, should the number of agent be greater, with the same number of observed ancestors, the worst case scenario probability would have been dramatically reduced. It is related to choosing *consecutive* sequences of m nodes among n. When n is large enough the number of such sequences is low compared to the overall number of random sequences, thus there is no need in practice to observe every ancestor.

Figure 6 shows the number of completed tasks and makespan of an application using DIET when using architecture failure recovery algorithm or not. In this experiment the DIET infrastructure is deployed as described in Figure 6(a). Client application submits a large number of independent `dgemm` calls. The `dgemm` DIET service is a remap of the BLAS DGEMM. Matrix size is set to 6000x6000, accuracy to 0.1 ; Average time to run a single job is 28.24s. An initial batch of 100 jobs is submitted. Then each time 10 jobs are completed 10 new jobs are submitted. We always inject failures following the same scenario: first a failure hits Paris cluster (loosing 16 computing nodes), then two simultaneous failures at Lyon (loosing 24 computing nodes), last one failure hits the root node of Lyon (loosing entire Lyon's deployment: 48 nodes). Compared to fault free execution, failures of 4 infrastructure nodes without recovery action leads to a major performance degradation. Most computing nodes are unavailable for computation due to only a few failures. When using the failure recovery algorithm, most failures are not even noticeable. The recovery algorithm gets back computing resources available faster than the grain of the submitted tasks, typically unavailable nodes are still running previously allocated jobs during architecture recovery. Though the third failure occurs when a slice of nodes have to be allocated. As a consequence no more free nodes are available and new jobs share computing power with previous ones, increasing execution time.

VIII. Concluding Remarks

We introduce three mechanisms intended to decrease the cost of failures in Network Enabled Servers environments. These mechanisms are implemented in the DIET GridRPC environment and their performance are evaluated experimentally in a large Grid System.

First we introduce a novel checkpoint interface to manage service recovery. It allows for using both automatic or, if available, service provided checkpoint. Parallel services can also be automatically recovered by cooperating with some automatic distributed checkpoint library. We provide a MPICH-V enabled service as a proof of concept. Checkpoint data are stored in a distributed grid persistent storage to recover from the complete loss of the computing resource, should it be a parallel cluster.

We also propose a distributed recovery algorithm eventually ensuring that all computing resources are kept available to the NES architecture. It can tolerate several simultaneous failures of agents, depending on the number of known ancestors kept by each agents. Our experiments shows a fast recovery time, indeed dependent on the failure detection time. Worst case scenarios where detection time stacks are unlikely to happen. Over a real grid deployment, most architecture recovery overhead is overlapped with computation, and the overall throughput of a typical BLAS based numerical application is greatly improved compared to a non fault tolerant architecture.

Finally our results outline the large impact of failure detection time, which is added to recovery overhead. We provide the first implementation and evaluation of the Chandra & Toueg & Aguillera failure detector in a grid system. It shows a very little overhead on network and memory and a nearly perfect accuracy when observing thousands of processes in real Grid.

Future works

The architecture recovery algorithm presented in this paper only keeps the hierarchy connected. It does not replaces failed agents, so overtime the scalability of the architecture might decrease. Next step is to restart agents to rebuild the same hierarchy. A first idea consists in using a centralized stable node to detect failures and restart failed nodes with the JADE [19] framework.

Most scheduling algorithms are relying on bandwidth performance prediction [20] to choose which server should be used to compute the request. Checkpoint data traffic may impact the available bandwidth. We plan to investigate the overall cost on application execution time and on scheduling accuracy of checkpoint data distribution on nodes.

During restart of a failed node, we request the DIET architecture to get a new resource to run the recovered job. Currently the standard scheduling algorithm is used. Restart time may be improved by scheduling the restarted job on a resource "close" to one of the replicated checkpoint holders. We plan to implement such a scheduling and to evaluate its merits.

References

[1] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Sagi, Z. Shi, and S. Vadhiyar, "Users' Guide to NetSolve V1.4," University of Tennessee, Knoxville, TN, Computer Science Dept. Technical Report CS-01-467, July 2001, http://www.cs.utk.edu/netsolve/.

[2] H. Nakada, M. Sato, and S. Sekiguchi, "Design and implementations of ninf: towards a global computing infrastructure," *Future Generation Computing Systems, Metacomputing Issue*, vol. 15, no. 5-6, pp. 649–658, 1999, http://ninf.apgrid.org/papers/papers.shtml.

[3] E. Caron and F. Desprez, "Diet: A scalable toolbox to build network enabled servers on the grid," *International Journal of High Performance Computing Applications*, vol. 20, no. 3, pp. 335–352, 2006.

[4] K. Seymour, C. Lee, F. Desprez, H. Nakada, and Y. Tanaka, "The End-User and Middleware APIs for GridRPC," in *Workshop on Grid Application Programming Interfaces, In conjunction with GGF12*, Brussels, Belgium, Sep. 2004.

[5] Y. Tanimura, T. Ikegami, H. Nakada, Y. Tanaka, and S. Sekiguchi, "Implementation of fault-tolerant GridRPC applications," in *Journal of Grid Computing*, vol. 4, no. 2. Springer Neth., june 2006, pp. 145–157.

[6] E. Caron, F. Desprez, F. Petit, and V. Villain, "A Hierarchical Resource Reservation Algorithm for Network Enabled Servers," in *IPDPS'03. The 17th Int. Parallel and Dist. Processing Symp.*, Nice - France, Apr. 2003.

[7] W. Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors," *IEEE Transactions on Computing*, vol. 51, no. 1, pp. 13–32, 2002.

[8] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10.

[9] H. Nakada, Y. Tanaka, S. Matsuoka, and S. Sekiguchi, "The design and implementation of a fault-tolerant RPC system: Ninf-c," in *Proceeding of HPC Asia 2004*. IEEE CS, 2004, pp. 9–18.

[10] A. Agbaria and J. Plank, "Design, implementation, and performance of checkpointing in netsolve," *dsn*, vol. 00, p. 49, 2000.

[11] S. Djilali, T. Herault, O. Lodygensky, T. Morlier, G. Fedak, and F. Cappello, "RPC-V: Toward fault-tolerant RPC for internet connected desktop grids with volatile nodes," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 39.

[12] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the condor distributed processing system," University of Wisconsin-Madison, Tech. Rep. 1346, 1997.

[13] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, pp. 494–499, 2006. [Online]. Available: http://stacks.iop.org/1742-6596/46/494

[14] G. Antoniu, J.-F. Deverge, and S. Monnet, "How to bring together fault tolerance and data consistency to enable grid data sharing," *Concurrency and Computation: Practice and Experience*, no. 17, 2006, to appear. [Online]. Available: http://hal.inria.fr/inria-00000987

[15] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V project: A multiprotocol automatic fault tolerant MPI," *International Journal of High Performance Computing and Applications (IJHPCA)*, vol. 20, no. 3, 2006.

[16] P. K. Chouhan, H. Dail, E. Caron, and F. Vivien, "Automatic middleware deployment planning on clusters," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 517–530, Nov. 2006.

[17] M. Schneider, "Self-stabilization," *ACM Computing Surveys*, vol. 25, no. 1, pp. 45–67, 1993.

[18] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," in *PODC '92: Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 1992, pp. 147–158.

[19] S. Bouchenak, F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, S. Jean-Bernard, N. de Palma, and V. Quema, "Architecture-based autonomous repair management: An application to j2ee clusters," in *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 13–24.

[20] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, "Steady-state scheduling on heterogeneous clusters," *Int. J. of Foundations of Computer Science*, vol. 16, no. 2, pp. 163–194, 2005.