Implementation of Fault-Tolerant GridRPC Applications

Yusuke Tanimura, Tsutomu Ikegami, Hidemoto Nakada, Yoshio Tanaka, and Satoshi Sekiguchi
National Institute of Advanced Industrial Science and Technology
Grid Technology Research Center, 1-1-1 Unezono, Tsukuba Central 2
{yusuke.tanimura, t.ikegami, hide-nakada, yoshio.tanaka, s.sekiguchi}@aist.go.jp

Abstract

In this paper, a task parallel application is implemented with Ninf-G which is a GridRPC system, and experimented on, using the Grid testbed in Asia Pacific, for three months. The application is programmed to run for a long time and typical fault patterns were gathered through tens of long executions. As a result, unstable network throughput was determined to be one of the biggest reasons for faults. Then, an important point for application developers is stressed, reminding them to avoid serious decline of task throughput during operations for faults, by timeout minimization for fault detection, background recovery and duplicate task assignments. This study also issues a steer for design of the automated fault-tolerant mechanism in a higher layer of the GridRPC framework.

1. Introduction

Recently, much has been expected of the Grid computing infrastructure in terms of large-scale computation for scientific discovery. A user will always have reasons to use a remote computer, but the Grid offers hope for scientists to be able to finish several-years-computation in several weeks. In addition, the Grid middleware technology that the Globus Toolkit[1] represents is being developed and matured based on past research. Indeed, various experiences on development and evaluation of Grid-enabled applications, such as a climate simulation[2], solving Einstein's equations[3], and etc., have been reported and they showed the possibility of the Grid as a ready and realistic infrastructure to be used by real science. There is, however, still a severe problem caused by unreliability of the Grid towards making the Grid a infrastructure for large-scale scientific applications.

Sudden faults and scheduled/unscheduled maintenance of networks, machines and software interrupt the promise of long running applications. For example, it is usual that a hard disk on a computing node in a cluster composed of more than 100 nodes will crash at least once in several months. A Grid-enabled application should be fault-tolerant and it is expected to have capabilities to detect faults appropriately and cope with the faults without terminating the entire execution of the application. In order to implement fault-tolerant applications, there is a need to develop applications and advanced middleware to overcome the instability of the Grid, but few discussions about what kinds of faults will happen and how middleware and applications can cope with those faults.

Several researches on fault-tolerant mechanisms have been reported, however the proposed scheme could be applied to a specific category of applications or immature for use by real applications. For example, Condor[4] implements checkpointing and process migration, however they are practically usable

Table 1. Major APIs and the corresponding error codes

API	Possible fault detected	Error code
grpc_function_handle_init()	DNS query failure	GRPC_SERVER_NOT_FOUND
	Server machine is down.	GRPC_SERVER_NOT_FOUND
	Network to server is down.	GRPC_SERVER_NOT_FOUND
	Globus-gatekeeper is not running.	GRPC_SERVER_NOT_FOUND
	GRAM invocation fails at authentication.	GRPC_OTHER_ERROR_CODE
grpc_function_handle_destruct()	–	–
grpc_call()	TCP disconnection during data transfer	GRPC_COMMUNICATION_FAILED
	RPC failure to disconnected server	GRPC_OTHER_ERROR_CODE
grpc_call_async()	TCP disconnection during data transfer	GRPC_COMMUNICATION_FAILED
	RPC failure to disconnected server	GRPC_OTHER_ERROR_CODE
grpc_cancel()	–	–
grpc_wait_any()	TCP disconnection (for nonblocking data transfer)	GRPC_SESSION_FAILED
	Hearbeat has timed out.	GRPC_SESSION_FAILED

only to a single job submission. Several works on fault-tolerant MPI have been reported[5, 6], however they are still in the research phase and still immature for use by real applications.

In this research, we investigated how to make Grid-enabled applications fault-tolerant. We have developed a task-parallel application with Ninf-G which is a GridPRC system[7], and made experiments on, using the Grid testbed in Asia Pacific, for three months. Our application implements simple fault-tolerant mechanism in order to run as long as possible. Through a long-term experiment, we analyzed fault patterns that actually happened and improve fault-tolerant functionality in the cost of fault detection and recovery operation, towards future support by the higher library of the GridRPC. In this paper, we summarize several notes in developing an application that needs a long run, and also indicate what functions middleware should offer. In Section 2, we will give an overview of the GridRPC and the Ninf-G library. Section 3 describes the implementation of a fault-tolerant application with Ninf-G. Details of the long-term experiment, experimental results and insight gained through the experiment are described in Section 4. Section 5 describes related works and Section 6 gives summary and future works.

2. Ninf-G design for fault-tolerance

Ninf-G is a reference implementation of the GridRPC. In this section, the basic concept of developing the GridRPC application, which has the fault-tolerance, using the Ninf-G, is introduced.

2.1 Overview of GridRPC

GridRPC[8], which is an RPC mechanism tailored for the Grid, is an attractive programming model for Grid computing. The programming model is that of standard RPC plus asynchronous, coarse-grained parallel tasking, in practice there are a variety of features that will largely hide the dynamic, insecure, and unstable aspects of the Grid from programmers. The GridRPC API has been proposed for standardization at the GGF (Global Grid Forum, http://www.gridforum.org) since 2003, in order to make the compatibility between several RPC-functional Grid systems.

Ninf-G[7], which is currently Version 2 and built on the Globus Toolkit Version 2, provide the GridRPC-based programming model to develop task-parallel applications easily. This work is performed with Version 2.2.0 of Ninf-G released on September 10, 2004. So far, a climate simulation[2] and a replica exchange Monte-Calro simulation[9] have been implemented to evaluate the basic performance and scalability of Ninf-G. However, there has been no deep analysis of fault-tolerant application development.

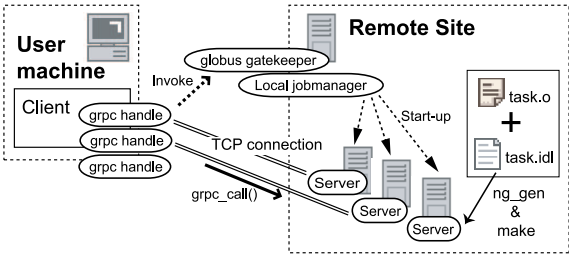

Figure 1. Connection between client and server of Ninf-G

Basically, GridRPC is a client-server model and it is assumed that some server processes are running on a remote resource to execute a specific routine. In developing a task-parallel application, both client and server programs are implemented separately and the client program calls the GridRPC function to carry out the remote routine. This paper describes ”server” as a Ninf-G server process and ”client” as a Ninf-G client process. Those terms don’t indicate a specific machine or computer resource.

Regarding the GridRPC programming and the Ninf-G behavior, a client creates several function handles each of which represents a mapping from a function name to an instance of that function on a particular server. A TCP connection between client and that server is established at this moment. Afterwards, all RPC calls using that function handle will be executed on the server specified in that binding. In the case of asynchronous RPC, the client can wait for a specific RPC using the Session ID given in the asynchronous call function. The Session ID is an identifier representing a particular asynchronous call and it can be utilized to cancel the call or check the status of the RPC. At the end of the program, all handles are released to end each server with the TCP disconnection. The client keeps those TCP connections during this flow, such as that shown in Figure 1. The connection must be established and maintained normally so that the application works. In other words, any faults will be detected on the client by looking at the status of all connections to the servers.

2.2 Fault-tolerance and GridRPC API

For application drivers, scheduled/unscheduled interruption for networks and machines cannot be avoided on the Grid environment when running their applications. An application program should continue to be processed in spite of that interruption, or it could be restarted midway. Although the middleware is expected to support applications that can do those things without complex operations required of the users, it has not been available at the production level, as described in Section 1. On the other hand, the GridRPC standard provides a primitive API set, and fault-tolerant functions should be supported in higher APIs. According to the GridRPC design, the primitive API detects any faults appropriately, and informs the application program rapidly. Specifically, the called GridRPC API should return a corresponding error code. In the case of that the error code will not be back, the heartbeat function provided by Ninf-G will be useful for an application program to avoid the deadlock.

2.3 Error handling

Table 1 shows major GridRPC APIs and the error codes that will be returned in encountering errors for the Ninf-G client. Discussion of the error codes is on-going at one of the working groups of GGF, but Ninf-G provides them based on the draft. As Table 1 shows, different kinds of faults correspond to the same error code, and it is impossible to know what happened or why the fault happened

just by reading the error codes. Because the error codes only show the status of the system, application developers can describe how their program should work, based on that status. For example, GRPC_COMMUNICATION_FAILED means there is a disconnection between client and server on the RPC route. After the error occurred, the handle is invalid until the client recreates the handle as it is programmed. The application developer of Ninf-G should describe in the client program when the client restarts the server, or where it does so.

2.4 Heartbeat function

Ninf-G provides a heartbeat function that sends a small packet periodically from server to client, in order to avoid disconnection by firewall software because there is no packet communication for a certain period of time. This function is also utilized to check if the server works without errors. If many packets are dropped on the network, this function will prevent the client from waiting for incoming data and being blocked forever.

An application user should set a timeout value to close the TCP connection if no packets arrive within the timeout seconds. If timeout has occurred, grpc_wait_any() would be returned with the GRPC_SESSION_FAILED code and the server handle would be invalid.

3. Implementation of TDDFT with fault-tolerance

A sample program reviewed in this study implements a TDDFT (Time-Dependent Density Functional Theory) equation[10]. TDDFT is a molecular simulation method in computational quantum chemistry, and it treats multiple electrons excitation directly, using techniques from quantum mechanics. An original program of TDDFT contains a hotspot to be processed in parallel using hundreds of CPUs, and the hotspot is iterated thousands of times to get an accurate result. This iteration requires a long-time execution. In our implementation, therefore, the hotspot is divided into several tasks that are executed in parallel. Figure 2 shows the simple flow of our TDDFT calculation. The flow consists of 1) Configuration of input data and parameters, 2) Multiple-tasks execution in parallel, using Ninf-G and 3) Calculation of the sequential part. 1) is only performed in the first loop and then 2) and 3) are repeated as times as a user wants.

In the Ninf-G application, a user should have responsibility for creating a server handle which is related with the server that the user wants to use. This insists on user-level implementation to manage the servers to which a task is assigned and to handle down servers for fault tolerance. For the long run of TDDFT, simple server management and task assignment methods are implemented including fault discovery and recovery operation, which are explained in this section.

3.1 Ninf-G server management and task assignment

As shown in Figure 2, each task of 2) must be processed after 1), which includes reading input data about molecules and setting given parameters. In this paper, the first RPC is named the ”initialization method,” and the second RPC is named the ”main method.” An array generated by the initialization method is maintained and can be referenced by the main method; that is a feature of the remote object provided since Ninf-G version 2. In particular, TDDFT requires the same initialization on all servers, before performing the main method. The simple flow of a TDDFT process is to initialize an array on all servers and then submit one task to them. A server that has finished the task will be given another task

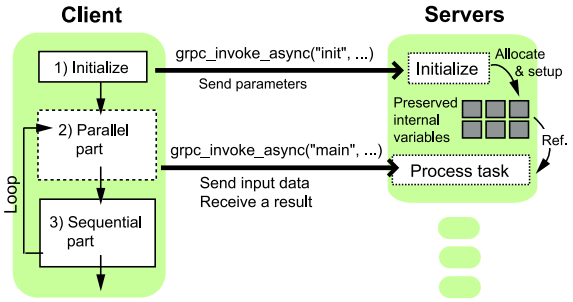

Figure 2. Implementation of TDDFT using Ninf-G

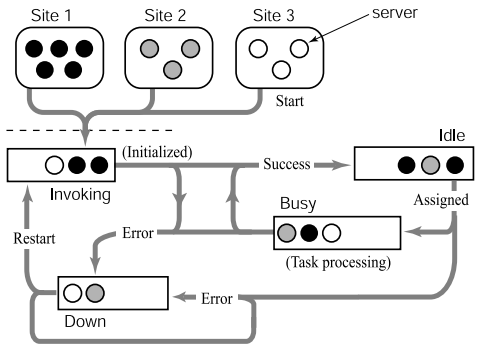

Figure 3. Status graph of the server

by the client. All tasks in each time step must be complete before the next step. On the basis of the flow, our program implements status management of all Ninf-G server processes as shown in Figure 3.

a) The status of a spawned server process is ”Invoking.”
b) The status moves from ”Invoking” to ”Idle” after completion of the initialization method.
c) A task is assigned to the ”Idle” server and its status is ”Busy” during calculation.
d) The status returns to ”Idle” after a task has finished without error.

3.2 Fault discovery and the recovery operation

In this paper, our motivation to implement fault tolerance is to enable a long-time execution on the real Grid. To achieve this goal, the client should not stop at any faults and the server which suffered the faults should be restarted when it becomes available again. Those two points are least requirements for our experiment and they are implemented in the application program.

When faults happen, the TCP connection between client and server is closed and the server process automatically exits. In the client, an appropriate error code is returned by the GridRPC system as described in Section 2.3. Therefore the client is programmed to change the status of the server which returns the error to ”Down.” Any tasks will not be assigned to ”Down” server any more.

For the server process to be restarted by the client, the client is programmed to destroy the server handle and create it after the fault is solved. Subsequently, the initialization method is called because the remote-object function is utilized in this case. When those processes are completed, the status of the server moves to ”Idle.” Since this TDDFT program requires the same initialization method on all servers and the calling parameters are static, it is possible to register the initialization method so that it is called again at recovery time.

A recovery operation is implemented to check the status of all servers every hour, find a handle array whose handles are all invalid, and recreate the handle array. The handle array is a set of handles on compute nodes which belong to the same cluster. The design of the handle array is based on the fact that the Ninf-G client requests the server process using GRAM (Grid Resource Allocation Manager) of the Globus Toolkit, and that the Globus gatekeeper also submits the server invocation to a local job manager, such as PBS or LSF. Because GSI (Grid Security Infrastructure) authentication of GRAM takes a certain overhead, the client will ask for multiple-servers invocation with one request with GRAM. Several nodes that are running a server managed by the same GRAM request will not be released even if one of them is down. The batch system is not suitable to restart only some Ninf-G servers among the servers managed

by one server handle array, because normal servers will also be restarted in vain. The recovery interval should be considered when figuring the cost of this operation.

4. Experiments

4.1. PRAGMA routine-basis experiment and its testbed

A developed TDDFT application was experimented as a PRAGMA routine-basis experiment. The Pacific Rim Application and Grid Middleware Assembly (PRAGMA)[11] is an institution-based organization, consisting of twenty-seven institutions around the Pacific Rim who are dedicated to building sustained collaborations and to advancing the use of Grid technologies in applications among a community of investigators working with leading institutions around the Pacific Rim. Applications are the focus of PRAGMA and are used to bring together the key infrastructure and middleware necessary to advance application goals.

Asia Pacific Partnership for Grid Computing (ApGrid)[12] is an open community encouraging collaboration. As of the end of May 2005, 49 organizations from 15 economic were participating in ApGrid. PRAGMA and ApGrid have been collaboratively building Grid testbed in Asia Pacific region.

PRAGMA routine-basis experiment was initiated in May 2004 and its purpose is to learn requirements and issues for testbed operations and developments of Grid-enabled applications through exercise with long-running sample applications on an international Grid testbed. PRAGMA routine-basis experiment was contacted by San Diego Supercomputer Center (SDSC) and TDDFT application is one of the applications used in the routine-basis experiment. One of our interests in this experiment is to watch actual fault patterns and discuss how to implement more appropriate fault-tolerance than tentative implementation. The experiments continued for 3 months, from June 1 to August 31 in 2004. For the routine-basis experiment, we provided a dedicated testbed which includes computational resources from AIST (Advanced Industrial Science and Technology, Japan), SDSC (San Diego Supercomputer Center, USA), KISTI (Korea Institute of Science and Technology Information), KU (Kasetsart University, Thailand), NCHC (National Center for High-performance computing, Taiwan), USM (University Sains Malaysia), TITECH (Tokyo Institute of Technology, Japan), NCSA (National Center for Supercomputing Applications, USA), UNAM (Universidad Nacional Autónoma México) and BII (Bioinformatics Institute, Singapore).

4.2 Results of the long run

The client program achieved a total of 906 hours (about 38 days) execution time during the experiment. The maximum number of sites that ran the Ninf-G server at the same time was 7 and the maximum number of CPUs was 67. The longest execution started at 20:00 JST on July 7, 2004 and continued until 16:00 JST on August 4, for a total of 164 hours (about 7 days). The execution environment of the longest run used 59 servers spread over 5 sites, as shown in Table 2. The Ninf-G client ran on the AIST node and it started the servers on remote sites using the server handle array. Throughput in Table 2 was measured by sending a 1 MB message from the AIST node to the remote-site node using TCP before the experiment. This TDDFT execution was able to divide the parallel part into 122 tasks in each loop. 4.87 MB of data was transferred from the client to the server on each RPC and 3.25 MB of data was transferred from the server to the client, as the result of the calculation.

Figure 4 shows the history of the live servers without the NCHC site. The reasons that the client recognized the server as ”Down” are as follows: 1) Data transfer failed at low network throughput, 2)

Table 2. Ninf-G server environment at the longest execution

Site	#CPU	System	Throughput
AIST	28	PentiumIII 1.4GHz	116 MB/s
SDSC	12	Xeon 2.4GHz	0.044 MB/s
KISTI	16	Pentium4 1.7GHz	0.28 MB/s
KU	2	Athlon 1GHz	0.050 MB/s
NCHC	1	Athlon 1.67GHz	0.23 MB/s

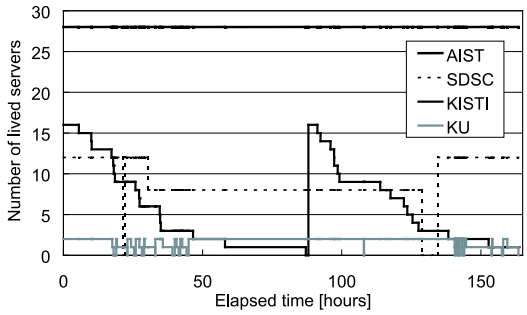

Figure 4. Transition of live servers

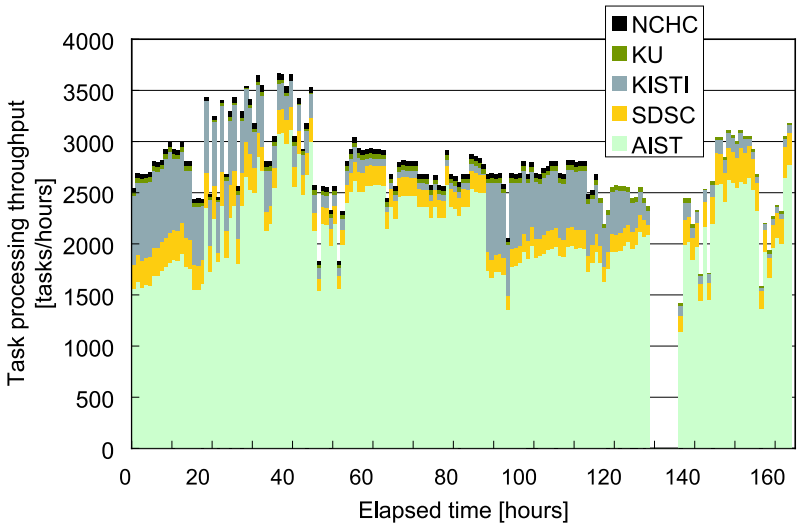

Figure 5. Transition of task throughput

The TCP connection was closed without notification to the client, and 3) The node was down because of heat problem or disk error. 1) and 3) were detected within short time by the TCP disconnection but 2) was only detected by the heartbeat timeout. Reasons identified as ”Down” servers were 67% for 1), 31% for 2), and 2% for 3) during the longest run.

Figure 5 shows the task throughput per hour. The reason that the task throughput was zero at 130 hours after start is that a boot request for the servers was queued for about 5 hours. As the execution time of one task was short, most tasks were processed on the AIST resource. If one of the remote sites was unavailable due to a fault, the task throughput tends to be higher. Nevertheless, the task throughput decreased during the time that the number of live servers was changing. Two reasons are considered as reasons for this problem. One is that the client program waits for all of the unfinished tasks each loop to be completed, and the resubmission caused by the fault might be the bottleneck. In addition, fault detection with the heartbeat takes from 300 to 400 seconds. The other reason is that the recovery operation for the ”Down” server is sequentially executed while task submission is pending. The solution is described in Sections 4.3, 4.4, and 4.5.

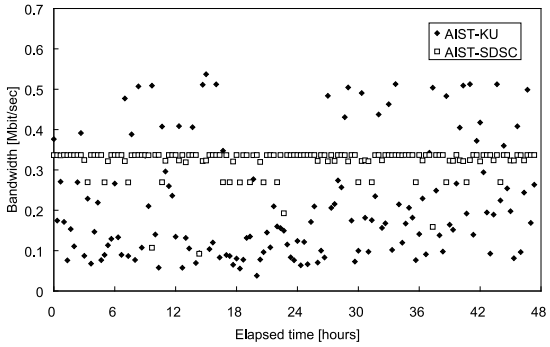

Figure 6. Transition of network throughput

Table 3. Heartbeat receiving interval and data transmission cost [sec]

	Heartbeat interval		Data transfer time	
Site	Ave.	Max.	Ave.	Max.
AIST	30.3	60.0	0.396	0.476
SDSC	30.6	81.0	19.4	25.7
KU	65.1	203	96.7	159

4.3 Stability of the network on the testbed

The KU node encountered faults more frequently than the SDSC node even when network throughput did not differ very much. Therefore, we investigated the details and the reason for the TCP disconnection. Figure 6 shows the network throughput of the AIST-SDSC path and the AIST-KU path every 20 minutes. The throughput of the AIST-KU path changed dynamically, and was sometimes very poor.

During 20 hours of TDDFT execution with 1 client on AIST, 4 servers on AIST and 4 servers on SDSC, the frequency of TCP retransmission from the client to the servers was 0.014 %. The same measurement with 1 client on AIST, 4 servers on AIST and 4 servers on KU showed 0.11 % as the retransmission ratio. The 8 times higher value also indicates that the TCP disconnection between AIST and KU was caused by an unstable network.

4.4 Improvement of fault detection

When the TCP socket was closed normally by a data transfer error of data transfer, etc., Ninf-G can detect the fault instantly. However, the detection will be delayed by at least the timeout seconds when the fault can only be found by the heartbeat timeout. But, if a user sets too small a number as the timeout value, a small delay will be judged as a fault and a normal connection may be terminated. We experimented how many seconds the heartbeat might be delayed by, in order to set an appropriate heartbeat timeout value.

Table 3 shows the receive intervals of the heartbeat with the program execution on 4 servers each on AIST, SDSC and KU. The servers on AIST and SDSC send a heartbeat packet every 60 seconds. The servers on KU send it every 80 seconds. If the client has not received any data or heartbeat by the time the interval has passed 5 times, that is, no data for 300 or 400 seconds, the RPC operation will be judged to be a heartbeat error. However, the heartbeat uses the same connection as the data transmission, so the heartbeat will not be received during data transmission. Instead of that, data arrival will be treated as a heartbeat reception. For this reason, the appropriate timeout value depends on how many seconds data transmission takes on average, and at the maximum, such as shown in Table 3.

In our experiment, the heartbeat, from SDSC and KU were delayed but the longest heartbeat arrival interval was shorter than the longest data transmission plus the heartbeat sending interval. Therefore, the total timeout seconds to give-up the RPC seems better to be set slightly longer than the longest data transmission, and shorter than the longest data transmission plus the heartbeat sending interval. On the other hand, it is effective that the heartbeat sending interval is shorter, and the allowed count of the

Table 4. Cost of the recovery operation [sec]

	AIST	SDSC	KU
1) Server halt	0.00709	0.00465	0.854
2) Server check	0.556	2.37	1.67
3) Server restart	4.83	10.6	2.80
4) Initialization	6.74	3.67	48.1
Total time	12.1	16.6	53.5

Table 5. Cost of the background recovery [sec]

No fault	215.62
Background recovery	216.26
Foreground recovery	263.35

heartbeat sending interval passed is large, in order to save on the cost of timeout wait.

4.5 Improvement of recovery operation

4.5.1 Recovery cost

From the results in Figure 5, it is revealed that the recovery cost affects task throughput. It is important to estimate the recovery cost appropriately and to minimize the cost by adjusting the timing or frequency of the operation. The cost of each recovery procedure was measured on 3 sites, AIST, SDSC, and KU and summarized in Table 4.

First of all, it is necessary to restart the server by releasing the server handle. In the case of starting the servers using the handle array, even not ”Down” servers are also restarted. The release of the handles to halt the server is shown in 1) in Table 4. 2) is used to request the globus gatekeeper to start the servers again. The client tries to get access to the globus gatekeeper, pass authentication, and request to the local scheduler. After all of these tasks are done, the server starts normally and notifies this to the client in 3). 4) is an operation used to call the initialization method to the server after 3). The cost of 4) depends on the particular application. The total cost includes operations 1) to 4).

The results shown in Table 4 represent the best value obtained out of 3 trials. The difference among 3 sites comes from divergence of processor speed, network performance, configuration of the local job manager and Globus, and etc. More cost to restart the KU servers might be required, depending on the network throughput. 3) and 4) take more time than 1) and 2) but those are server-side operations. The client should be processing other things while 3) and 4) are going on so that the costs of 3) and 4) are hidden from the client. If the recovery failed due to continuous network problems, etc., an error will occur in 2). 2) can be summed up as many retries of the recovery, but it can also be hidden by background operations with thread programming.

4.5.2 Background recovery

On the basis of the discussion in the last section, the background recovery was implemented with one thread that performs operations 1) to 4). For evaluation purposes, 2 sets of 6 servers were started on the AIST node to avoid unstable network performance and heterogeneity of computer performance. One of the sets was halted in the first loop during task processing, and the recovery operation was done after the loop. The foreground recovery process sequentially performs 4) for each server and waits for the recovery before the sequential part of TDDFT. The background recovery is moving on to the sequential part of TDDFT while the recovery is being tried. Table 5 shows how many seconds the calculation took for the sum of the first loop and the next. It was found that the background recovery took almost the same time as execution without the halt.

4.6 Discussion

In this fault-tolerant application development for a long run, the fault detection is implemented by catching the error code of the GridRPC API. Then, subsequent process is cancelled, a failed job is re-submitted to another server, and an affected server is recognized as ”Down” on the error condition. Nevertheless, it is not possible to detect the situation that a client freezes by waiting for unreceived packets for all time. The heartbeat function is necessarily enabled in execution to exit the trouble. In the experiment on the ApGrid/PRAGMA testbed, most faults were caused by decline of the network throughput. We learned the following points from our long-run experiment in order to make a sophisticated fault-tolerance mechanism into the development and execution of the GridRPC based application:

- The heartbeat function is useful to detect the fault on an unstable network. To improve usability of this function, it is expected to configure the timeout value automatically because the best value is not easy to be determined statically and easily.
- In case that the initialization method takes time, the cost of the server recovery will be large and processed in background. This issue is obvious but users should care more when a task requires initialization. In addition, the cost of the initialization method depends on the application.
- There is no method to determine how long the fault might continue. Periodic trials of the recovery cost a certain time, but it is revealed that the cost will not be large as a proportion of the total time. Naturally, a background trial is expected in implementation of the higher-level of GridRPC API.
- There is an issue with how some ”Down” servers can be treated by the same handle arrays as other normal servers, and can be restarted. A tradeoff exists between the restart cost of normal servers in vain, and no use of the servers that can be used at this restart. This should involve some functions of the local scheduler.
- When the remaining tasks are few, duplicate task assignment is very effective because failure of the final task increases waiting time due to the need for recalculation. Our results showed serious decline of task throughput without duplicate assignment.

5. Related work

There are other middlewares such as Condor MW[13] and Ninf-C[14] based on Condor, that implement a similar function to RPC. They achieve the checkpoint and the task migration at the middleware level, and an on-going-task can be restarted from some midpoint on another server. Netsolve[15], another implementation of GridRPC, provides automatic resubmission of a failed task to another server but no restart from a midpoint. In contrast to those middleware suites, Ninf-G does not provide a specialized information server that can treat a Ninf-G task. Ninf-G is designed such that fault-tolerance and server management should be implemented in the application level or in the higher middleware. Our work is important from the viewpoint of discussing fault-tolerant application development using the primitive GridRPC API. This will bring higher functionality of the GridRPC.

The faults we met in this experiment depend on the current Ninf-G that attempts to keep the TCP connection between client and server. The reason for keeping the connection is to obtain better performance for fine-grained task parallel processing. This paper does not focus on the issue of the long establishment of a TCP connection. NetSolve, which does not keep the connection, will not have an effect on the client program so much as against a network fault. However, the discussion about the failure of the heartbeat or data transmission is still necessary. The lessons learned from our experiment are useful for long-run applications to save costs in fault detection and server recovery.

6. Conclusion and future works

In this paper, several important points to be considered in the development and execution of a task-parallel application that can be run for a long time on the Grid have been discussed. Our client program was simply implemented with consideration to the error handling of the GridRPC for fault detection and management of the Ninf-G server to assign a task or to recover down servers. The recovery operation supported the remote object, too. Through the routine-basis experiment, typical fault patterns and ratio were inspected. Then our tentative implementation was improved for efficient resource utilization. In particular, minimizing of timeout-based detection, background recovery, and duplicate task assignment in advance were rewarding. However, they are plainly complicated in implementation and could be reproduced by several application users. They should be implemented in middleware and the result in this paper is a signpost to the design of a higher-level interface of the GridRPC and an increase in its functionality. In the new interface, we plan to design an easier function to submit multiple tasks one time over the primitive GridRPC APIs, including automatic and quick server recovery.

In addition, this paper raises an issue of faults at remote side, while the client is a single point of failure and it is assumed that users choose a reliable machine to run their clients. As our future works, restart of the client, rollback and resubmission of the task which was interrupted by client's fault, should be designed in the higher layer over the GridRPC.

Acknowledgement

We would like to thank to Dr. Nobusada (Institute of Molecular Science) and Dr. Yabana (Tsukuba University) who provided their TDDFT program for our research work. Also, we would like to give thanks to Mr. Takemiya (AIST) who gave us many comments on our work.

This work has been conducted as a PRAGMA routine-basis experiment and supported by PRAGMA and ApGrid communities. We are grateful for their resource contribution to our long term experiment. We would especially like to give thanks to Ms. Cindy Zheng (SDSC) who gave great leadership for conducting the routine-basis experiment.

A part of this research was supported by a grant from the Ministry of Education, Sports, Culture, Science, and Technology (MEXT) of Japan through the NAREGI (National Research Grid Initiative) Project.

References

[1] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Supercomputing Applications and High Performance Computing*, vol. 11, no. 2, pp. 115–128, 1997.

[2] H. Takemiya, K. Shudo, Y. Tanaka, and S. Sekiguchi, "Constructing Grid Applications Using Standard Grid Middleware," *Grid Computing*, vol. 1, pp. 117–131, 2003.

[3] G. Allen and et al., "Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus," in *Proceedings of Supercomputing 2001*, 2001.

[4] J. Pruyne and M. Livny, "Managing Checkpoints for Parallel Programs," 1996.

[5] E. F. Graham and et al., "HARNESS and fault tolerant MPI," *Parallel Computing*, vol. 27, pp. 1479–1496, 2001.

[6] G. Bosilca and et al., "Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes," in *Proceedings of Supercomputing*, 2002.

[7] Y. Tanaka, H. Takemiya, H. Nakada, and S. Sekiguchi, "Design, implementation and performance evaluation of GridRPC programming middleware for a large-scale computational grid," *Fifth IEEE/ACS International Workshop on Grid Communting*, pp. 298–305, 2005.

[8] K. Seymour and et al., "Overview of GridRPC: A Remote Procedure Call API for Grid Computing," in *Proceedings of 3rd International Workshop on Grid Computing* (M. Parashar, ed.), pp. 274–278, 2002.

[9] T. Ikegami and et al., "Accurate Molecular Simulation on the Grid – Replica Exchange Monte Carlo Simulation for c_{20} Molecule," *Journal of Information Processing Society of Japan*, vol. 44, no. SIG11, pp. 14–22, 2003.

[10] K. Yabana and G. F. Bertsch, "Time-Dependent Local-Density Approximation in Real Time: Application to Conjugated Molecules," *Quantum Chemistry*, vol. 75, pp. 55–66, 1999.

[11] "PRAGMA." http://www.pragma-grid.net/.

[12] "ApGrid." http://www.apgrid.org/.

[13] J. Goux, S. Kulkarni, J. Linderoth, and M. Yoder, "An Enabling Framework for Master-Worker Applications on the Computational Grid," in *Proceedings of HPDC-9*, pp. 43–50, 2000.

[14] H. Nakada, Y. Tanaka, S. Matsuoka, and S. Sekiguchi, "The Design and Implementation of a Fault-Tolerant RPC System: Ninf-C," in *Proceedings of HPC Asia*, pp. 9–18, 2004.

[15] H. Casanova and J. Dongarra, "Netsolve: A Network Server for Solving Computational Science Problems," *Supercomputing Applications and High Performance Computing*, vol. 11, no. 3, pp. 212–223, 1997.