Implementation of Fault-Tolerant GridRPC Applications

Workshop on Grid Applications

In conjunction with GGF 14, June 27, Chicago, USA

Yusuke Tanimura, Tsutomu Ikegami, Hidemoto Nakada

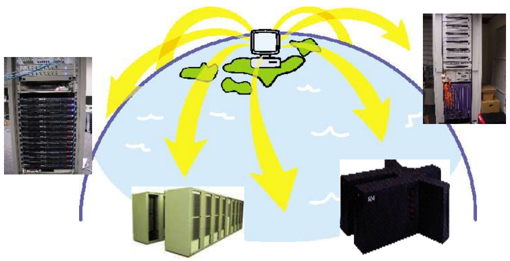
Yoshio Tanaka, Satoshi Sekiguchi

Grid Technology Research Center

AIST, Japan

Background

- **A large-scale computation for a long time is getting realistic because of recent improvement of middlewares.**
  - In our experiences at SC'03, SC'04
    - Climate Simulation : 500 processors
    - QM/MD: 2000 processors
  - A long time execution is also required from application side.
    - We might meet some faults during execution.
    - Many discussions about fault-tolerance are going on.
      - A few project like Condor showed a good achievement.
    - Issues for developing a FT application:
      - How should the application be programmed?
      - What functions should be offered by the middleware?



- **We have studied GridRPC for several years and develop Ninf-G as a reference impl.**

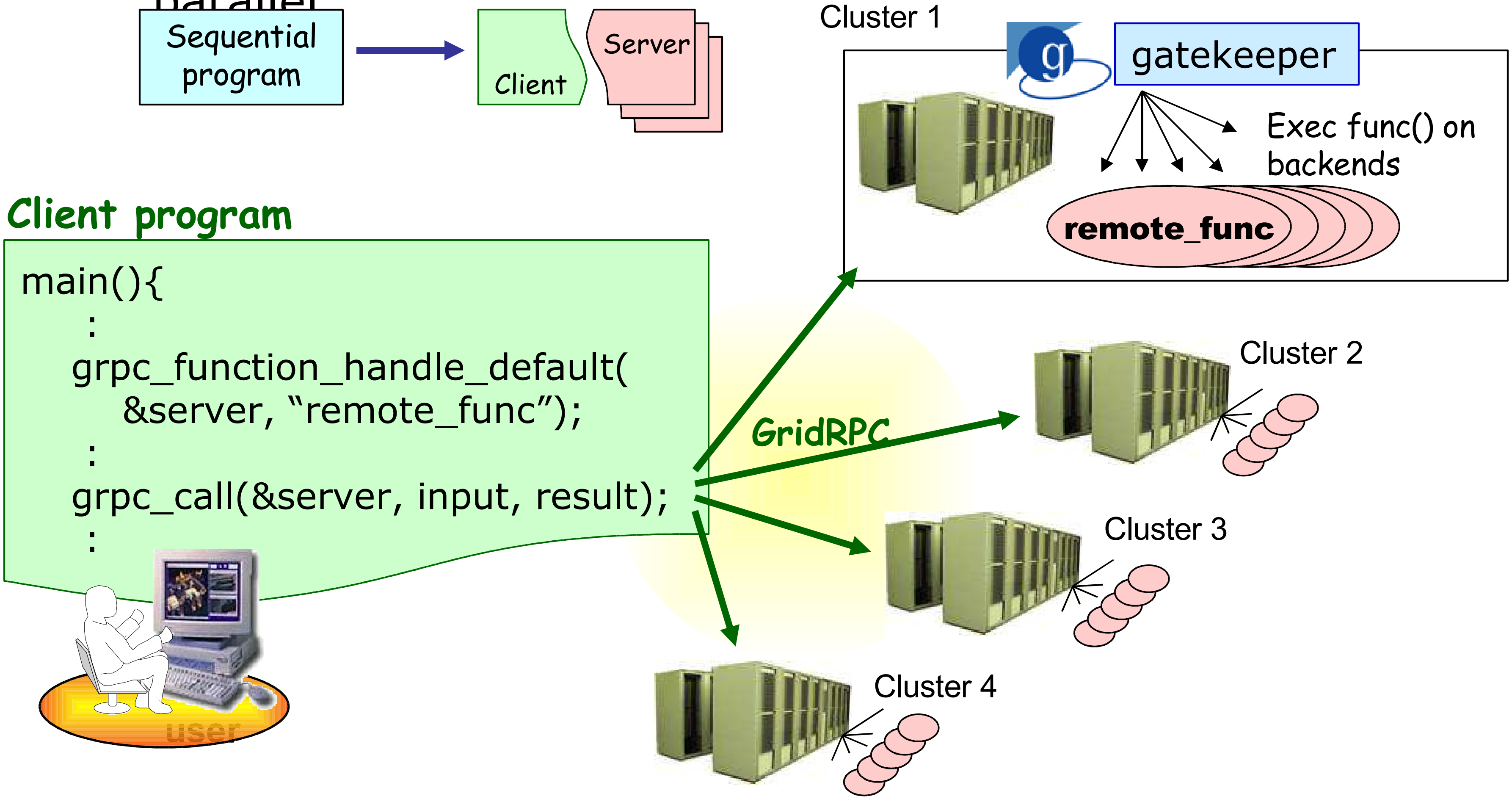
Purpose of this study

- **To run an application for a long time and analyze faults on an international Grid**
- **To clarify how to implement a fault-tolerant GridRPC application**
- **To reveal functions to be provided by the GridRPC middleware in the future**

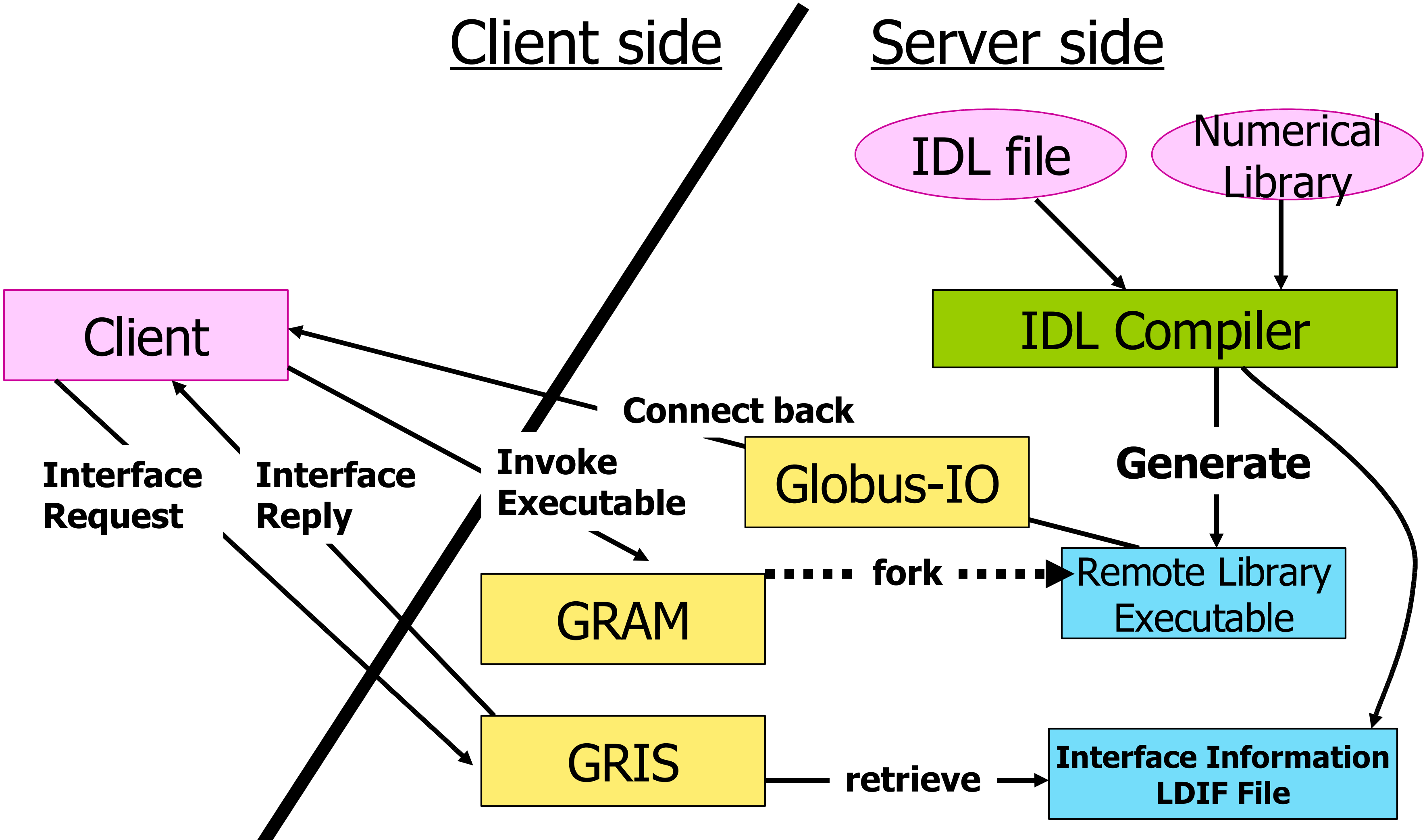
  - Today's agenda
    - Introduction of the GridRPC programming
    - Approach for making an application fault-tolerant
    - Result of a long time execution on the testbed
    - Notes for GridRPC middleware developers and application programmers

Overview of GridRPC

- **One of the programming models**
  - Execute partial calculations on multiple servers in parallel



Architecture of Ninf-G



Sample Code of GridRPC Client

Invoke Ninf-G servers

Submit a task to each active servers

Submit another task to the server which finished the task

Halt Ninf-G servers

```
    :
grpc_object_handle_array_init(handle)
    :                          → Get a function handle
for(i=0; i < N; i++)
   grpc_invoke_async(&handle[i], A);
    :                          → Call non-blocking RPC
while(1){
  grpc_wait_any();
    :                          → Wait for task completion
  grpc_invoke_async(&handle[x], A);
    :
}
    :
grpc_object_handle_array_destruct(handle)
    :
```

TCP connections between client and server are maintained during this period

Direction of Implementing FT

- **Execute our application as long as possible**
  - Along with the routine-basis experiments (Daily use of the Grid) of PRAGMA project
  - On the Asia Pacific Grid testbed operated by PRAGMA / ApGrid project
    - Unstable network in the Asia, Less practical experiments
    - What kinds of faults happens? How often?
  - Repeat the execution while improving the program

- **Development issues**
  - Application should continue calculation without down servers. A failed RPC should be performed on another server on another lived node.
  - Down servers should be restarted after a fault is resolved.

Make Application Fault-Tolerant

Design basis of the GridRPC

- End-user API to be released is a primitive API set.
- Task scheduling or fault-tolerant function should be built over the primitive API set.

Application-level implementation

- Catch every error code of the GridRPC API
  - Detect RPC failure and TCP disconnection
- Avoid dead lock of the client program
  - Use timeout mechanism
    - Ex. Server invocation, RPC session, heartbeat receive
- Manage status of Ninf-G servers for task assignment
  - Manage the function handle (that is corresponded to the server)
    - Handle of the active server or the inactive

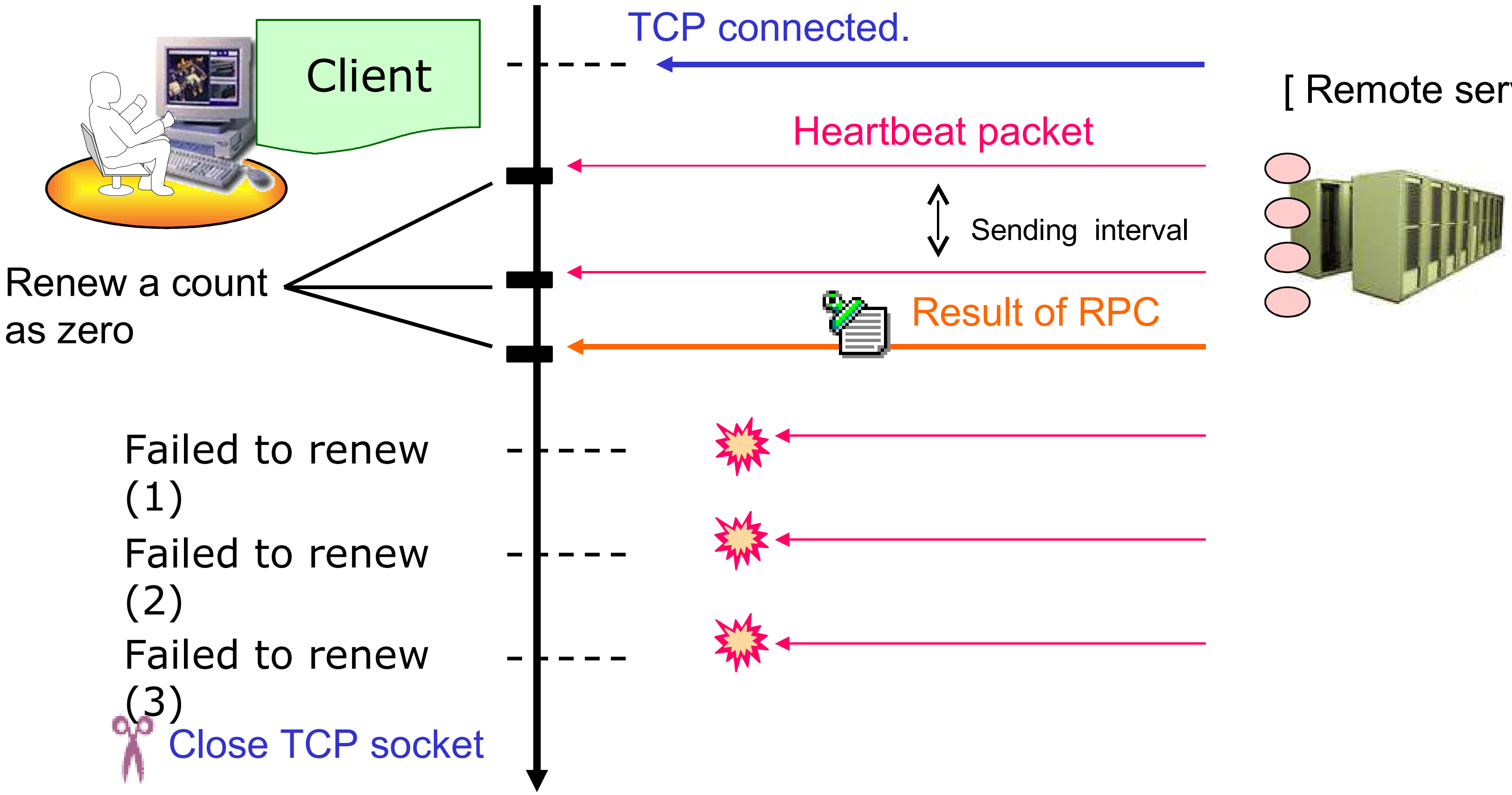
Error Codes of GridRPC APIs

[Major GridRPC APIs]	[Detectable Errors]
Invocation of the remote server grpc_[function | object]_handle_*()	→ No such host Cannot get access to the host → GSI authentication failure
RPC grpc_[call | invoke]() grpc_[call | invoke]_async()	→ TCP connection is closed. → Blocking data transfer failed. → RPC timed out.
Wait for non-blocking RPC requests grpc_wait*()	→ Non-blocking data transfer failed. → Heartbeat timed out.

Timeout mechanism

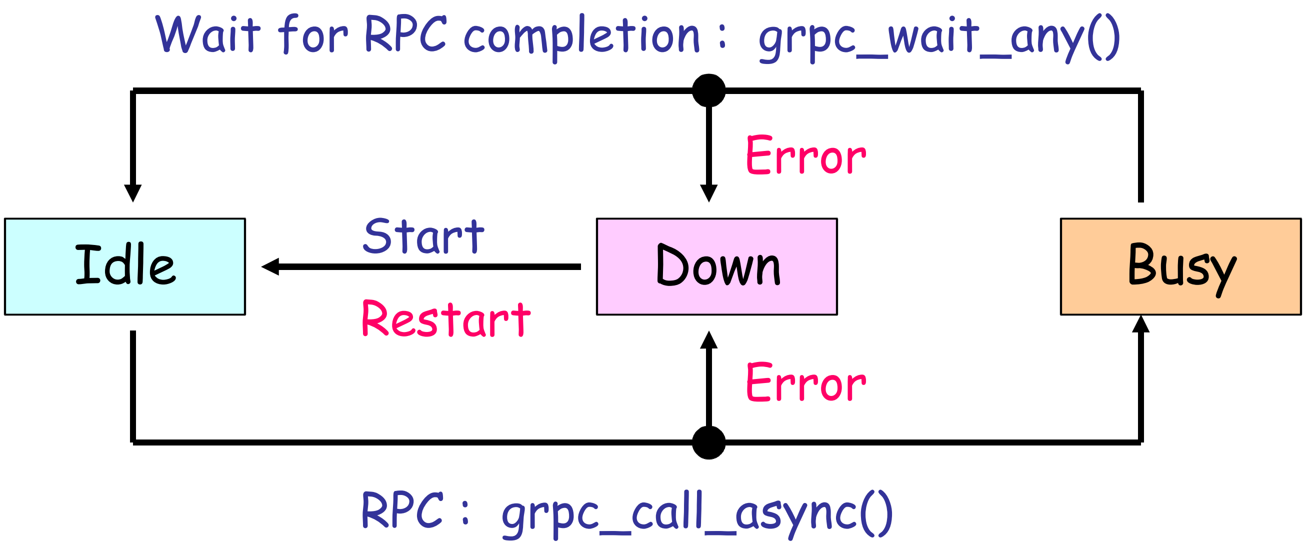
Example of heartbeat

- Users' configuration: Sending interval = 60 sec, Max count = 3
  - Timeout seconds will be 60 x 3 (= 180).



Servers' Status Management

- **Manage servers with 3 status**
  - Down, Idle, Busy (Tasking or Initializing)
- **Recovery operation**
  - The operation is applied to each handle array which all handles are inactive (that means servers are down).
  - The operation is performed once an hour.



Example Application (TDDFT)

- **TDDFT: Time-Dependent Density Functional Theory**
  - By Nobusada (IMS) and Yabana (Tsukuba Univ.)
  - Application of the computational quantum chemistry
  - Simulate how the electronic system evolves in time after excitation

Time dependent N-electron wave function is

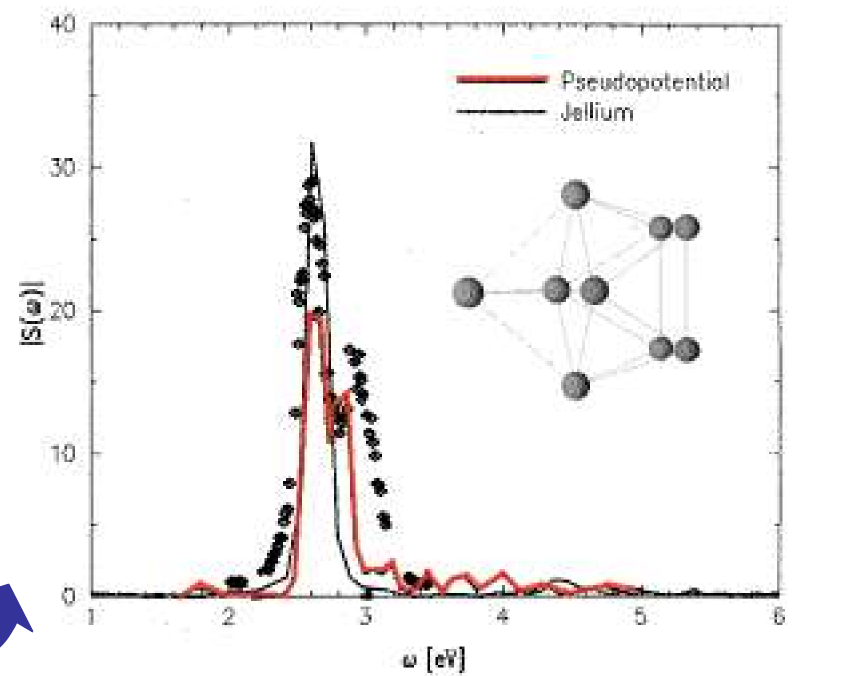
$$\left|\Psi\right\rangle = \left|\psi_1\psi_2\cdots\psi_N\right\rangle$$

which is approximated and transformed to

$$i\frac{\partial}{\partial t}\psi_i = \left(-\frac{1}{2}\nabla^2 + V_{ion} + V_H + V_{ex}\right)\psi_i$$

then applied to numerical integration.

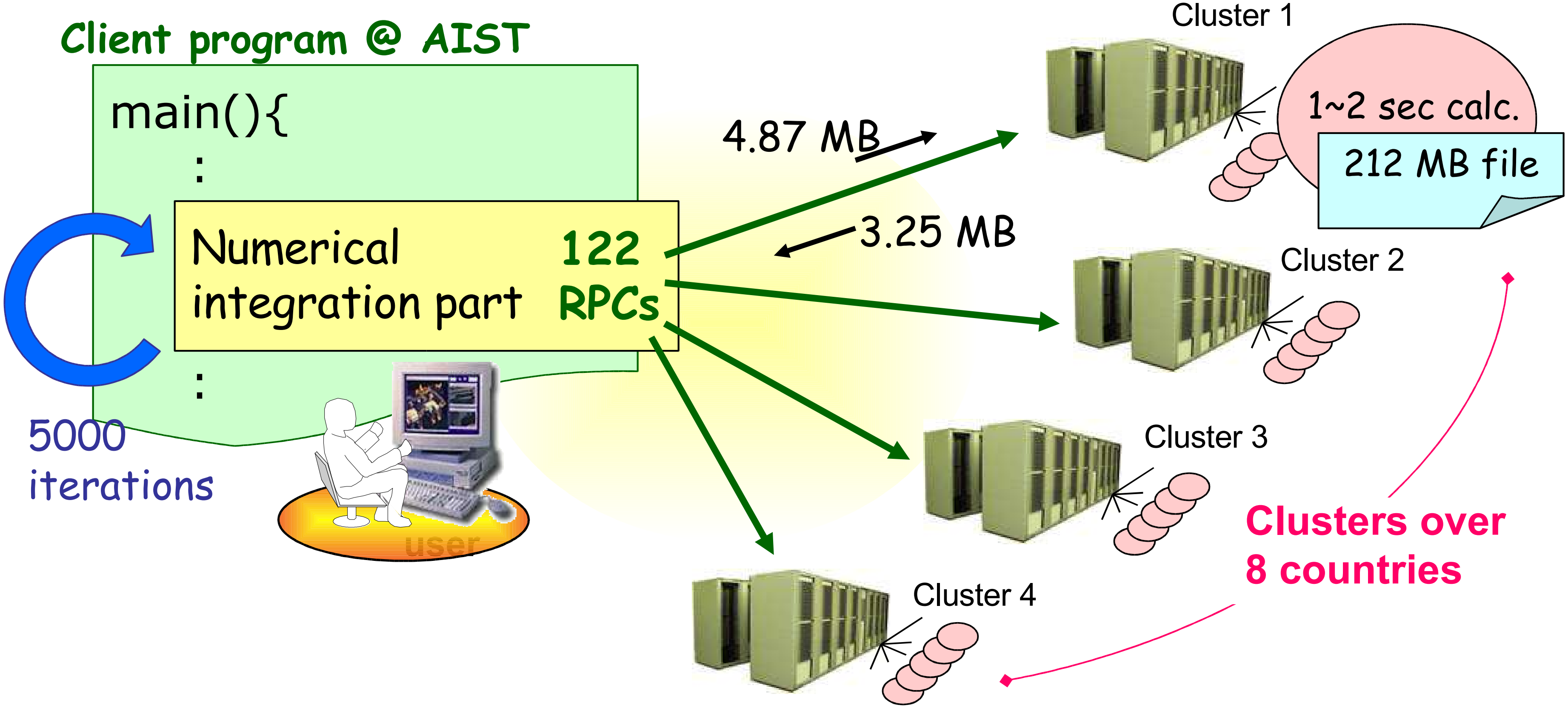
A spectrum graph by calculated real-time dipole moments



(F. Calvayrac, P.-G. Reinhard, and E. Suraud, J. Phys. B 31, 1367 (1998))

Parameters in Execution

- **Simulation of the legand-protected Au_{13} molecule**
- **Approximately 6.1 millions' RPCs are required (Take more than 1 week with 1 PC).**
  - Require high-throughput communication



PRAGMA/ApGrid Testbed

- **Total 8 countries / 10 sites / 104 nodes / 210 CPUs**

SDSC, USA
64 CPUs

AIST, Japan
28 CPUs

UNAM, Mexico
6 CPUs

TITECH, Japan
8 CPUs

NCHC, Taiwan
16 CPUs

KISTI, Korea
16 CPUs

KU, Thailand
16 CPUs

USM, Malaysia
32CPUs

BII, Singapore
16 CPUs

NCSA, USA
8 CPUs

Statistical Results for 3 months

- **Cumulative results**
  - # of executions by 2 users: 43
  - Execution time (Total) : 1210 hours (50.4 days)
    (Longest) : 164 hours (6.8 days)
    (Average) : 28.14 hours (1.2 days)
  - Total # of RPCs : 2,500,000
  - Total # of RPC failures : 1,600
    - Error ratio : 0.064 %
- **Major faults**
  - Unstable networks between client and server
    - Packet drop, Fluctuating throughput, TCP disconnection
  - Server node down
    - Due to heat, electricity, HDD and NFS problem, and moving

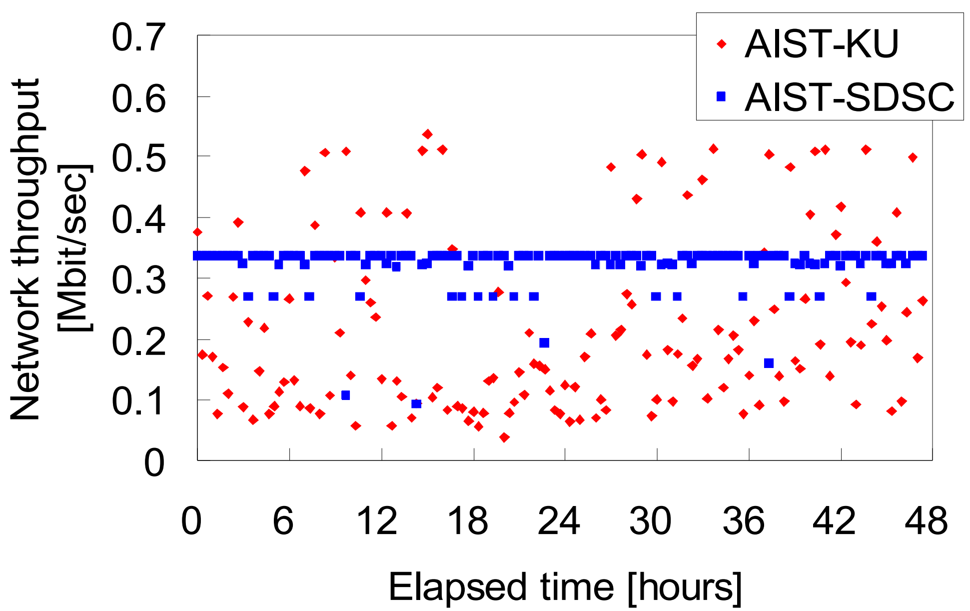
The Longest Run (Environment)

Site (Country)	#CPU	CPU	Throughput
AIST (Japan)	28	P3 1.4 GHz	928 Mbps
SDSC (USA)	12	Xeon 2.4 GHz	0.352
KISTI (S.Korea)	16	P4 1.7 GHz	2.24
KU (Thailand)	2	Athlon 1GHz	0.400
NCHC (Taiwan)	1	Athlon 1.67 GHz	1.84

AIST-KU network

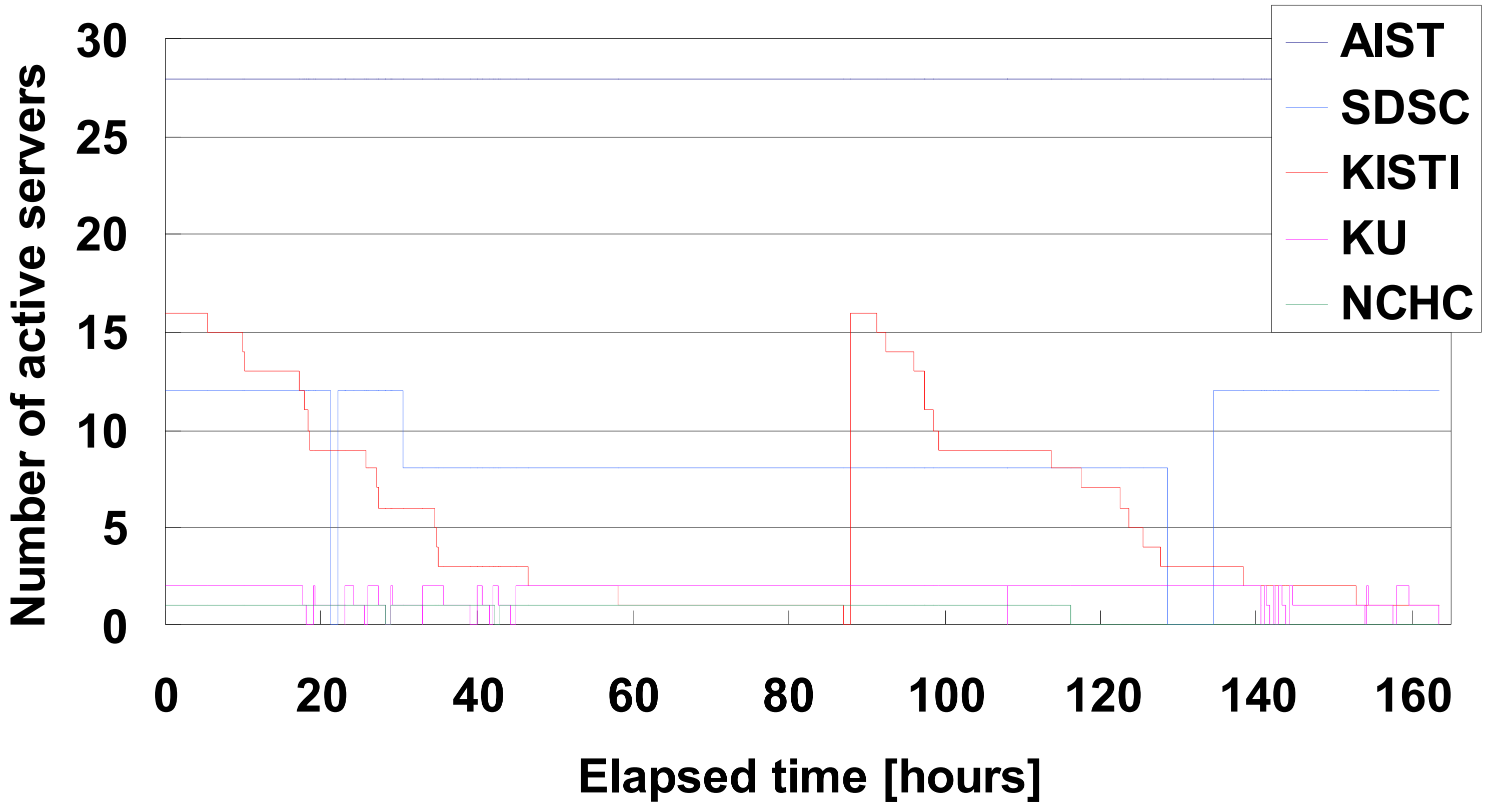
- Unstable throughput
- 8 times higher packet-loss ratio

NWS measurement
Send 16KB data 4 times
with 32KB socket buffer



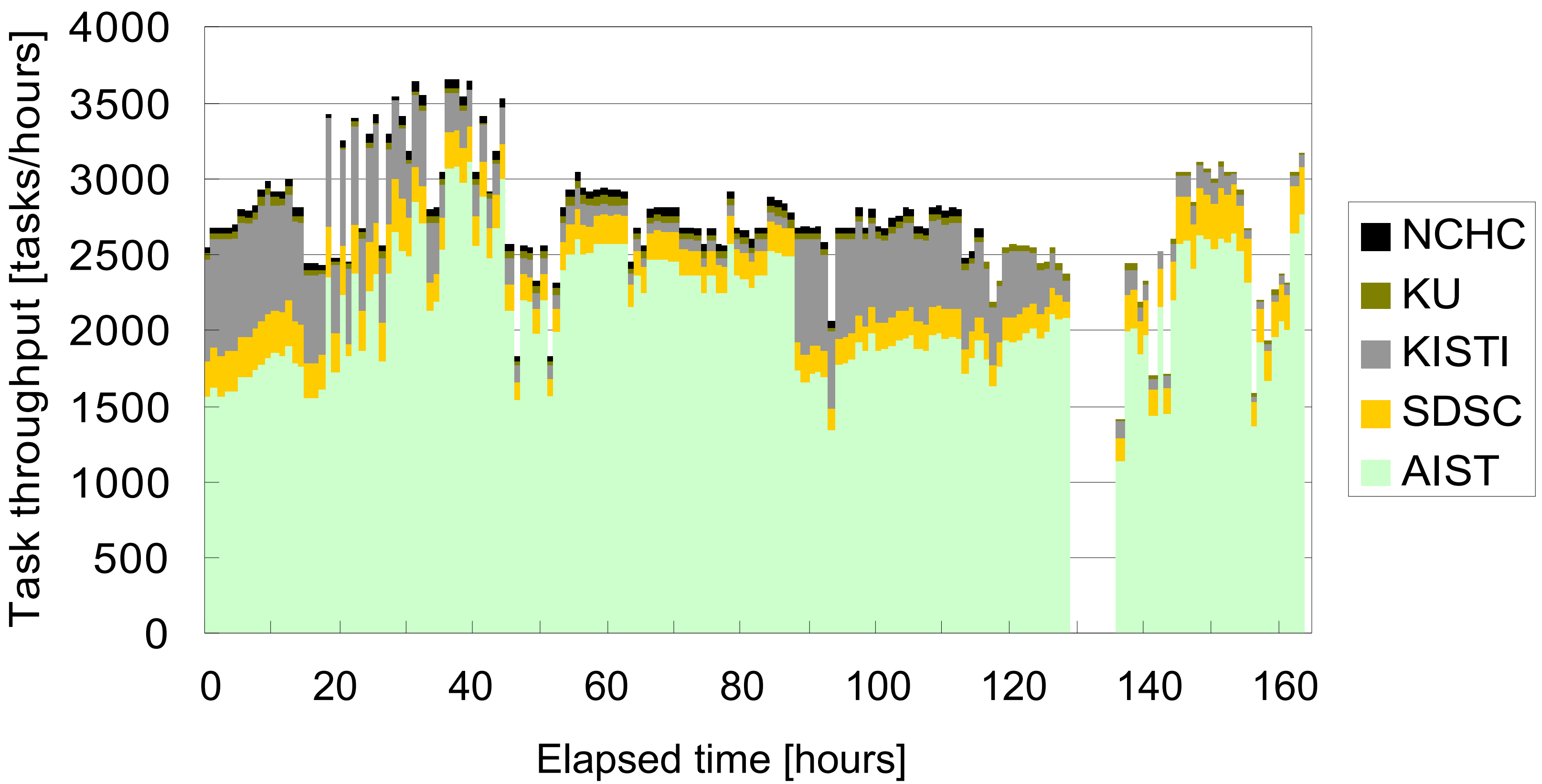
The Longest Run (Availability)

- **67 % : Data transfer failed at the sharp fall of throughput**
  - Socket closed (91.7 %), Misjudged timeout (8.3 %)
- **31 % : Timeout due to hung-up connection**
- **2 % : Node down**



The Longest Run (Task Throughput)

- **Low task throughput while detecting faults**
  - Not easy to set an appropriate timeout value for detection
- **Low task throughput when the final task failed**



Notes for developers (1/2)

- **For quick detection**
  - Some faults were detected only by timeout mechanism.
  - Hard to set an appropriate timeout value
    - Trade-off between detection speed and frequency of wrong detection
      - Timeout seconds will take for fault detection at the worst case.
      - Heartbeat packet is not sent during data (RPC result) transfer but it is not easy to predict how long the transfer takes.
  - Current use of timeout mechanism, in Ninf-G
    - Heartbeat, RPC session, server invocation and server halt

- **For quick recovery**
  - Recovery operation should be performed in background.

For Quick Recovery

- **Recovery operation flow**
  1. **Stop servers if alive**
  2. **Start servers with GRAM request**
  3. **Receive notification from "started servers" and establish TCP connection**
  4. **RPC of "initialization method"**
  5. **Put "initialized server" to Idle pool**

These operations of each cluster is independent and can be processed in background.

[Restart costs] *Client was running on the AIST node.

Site of server	AIST	SDSC	KU
1) Halt server	0.00709	0.00465	0.854
2) GRAM Auth	0.556	2.37	1.67
3) Boot server	4.87	10.6	2.80
4) Initialize	6.74	3.67	48.1

[seconds]

Possibly large

→ Depend on queue configuration

→ Depend on application

Notes for developers (2/2)

- **Duplicate task submission for high task-throughput**
  - Completeness of the final task is deterministic.
  - If the final task fails, execution time will seriously increase due to the need for recalculation of it.

- **Lack of cooperation between Ninf-G and the local batch system**
  - Impossible to restart some of Ninf-G servers which are submitted by the single “qsub” command through the GRAM request
  - On the other hand, “handle array” is only solution to boot hundreds of Ninf-G servers and keep them in stable.

Summary

- **Implemented a fault-tolerant application using GridRPC (Ninf-G)**
  - Our approach
    - For fault detection
      - Catch the error codes of the GridRPC APIs
      - Use timeout mechanism for fault detection
    - For fault avoidance
      - Manage status of the servers and tasks
    - For recovery
      - Restart servers in straightforward way
  - Experiment on the testbed in Asia Pacific
    - Analysis of faults
    - Discuss detection and recovery method
- **Revealed notes for the GridRPC middleware developers and application programmers**