P3: P2P-based Middleware Enabling Transfer and Aggregation of Computational Resources

Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi

Grid Technology Research Center
National Institute of Advanced Industrial Science and Technology
AIST Tsukuba Central 2, Tsukuba, Ibaraki, 305-8568, Japan
shudo@ni.aist.go.jp, {yoshio.tanaka,s.sekiguchi}@aist.go.jp

Abstract

This paper presents middleware enabling mutual and equal transfer of computing power between individuals, as in the original idea behind P2P, while also supporting large-scale distributed computation utilizing heterogeneous PCs. This goal is strongly supported by a network overlay over which peers can communicate with each other directly and bidirectionally. We made use of a general-purpose P2P library, JXTA, supporting the common requirements of P2P software, including network overlay. Other features of the P2P library, such as ad-hoc self-organization, discovery and grouping of peers, also support our middleware efficiently. In this paper, we propose and evaluate an application of those P2P concepts to virtual resource transfer and parallel computation with aggregated resources. However, such a P2P library imposes a certain amount of overhead on the middleware in terms of communication performance. Measured communication performance and throughput of an application program shows the feasibility of the application of P2P concepts. The middleware achieves 100×10^6 *bps communication performance and over a 20 fold increase in speed with 32 computers, even though the granularity of workunits is as fine as less than a second.*

1 Introduction

Recently, activities have emerged that allow us to use computational resources greater than we keep and maintain ourselves. In these circumstances, we can use others' resources and pay-per-use for the supplied services. Application Service Providers (ASP) offers such computational services and Utility Computing should enable transfer and trade of those services.

In Grid research and experiments, large-scale scarce resources, such as high-end supercomputers, data storage, observation sensors and experimental devices, have been transferred virtually, and shared mutually and securely across multiple organizations.

Internet-wide distributed computing projects [1, 2] are also resource transferring activities, but specialized for utilization of the computing power of individual PCs. In those projects, participants can only donate their resources. A participants can be only a "resource provider" and the project owner is the only "resource user." The two are asymmetric in capability. Such asymmetry is contrary to the the original idea of P2P, in which peers ought to have equal capability. Equality in resource transfer means that everyone not only provides his/her resources to others but also uses others' resources.

This paper presents middleware named P3 (Personal Power Plant) which we designed and developed. It enables mutual and equal transfer of computing power between individuals as called for in the original idea of P2P. Any user of P3 can use any others' resources by submitting a computational job. It also supports traditional large-scale distributed computation utilizing heterogeneous computers.

On the Internet of today, it is the normal situation that communication targets and directions are restricted. Firewalls and NA(P)T impose such constraints on most computers. However, it is highly desirable that any computer can initiate communication to other computers to achieve mutual transfer of resources. These constraints also restrict communication patterns and programming models in parallel processing to the master-worker model, and the like.

We made use of a general-purpose P2P communication library which provides a network overlay to get rid of the constraints. On the overlay any computer can communicate directly with other computers even over an underlying restricted network with firewalls. Most P2P software needs that sort of overlay and there have been development efforts, including a relay-based approach, and a hole-punching approach. By re-using these results, our middleware can benefit from improvement of the existing efforts.

Such a P2P library provides other benefits, such as ad-hoc self-organization of computers, discovery, and grouping of computers. These functions can efficiently support P2P distributed computation middleware, including P3. In this paper, we propose an application of

Proc. Cluster Computing and Grid 2005 (CCGrid 2005, Fifth Int'l Workshop on Global and Peer-to-Peer Computing), May. 2005.

these functions to this kind of middleware.

A P2P library undoubtedly provides us a rich set of features. But it is also certain that such a library introduces performance overhead. We confirmed the feasibility of a P2P library, JXTA, as a base for distributed computation by measuring basic communication performance and application throughput. These experiments also showed the current scalability of P3 in number of processing computers, and the workunit processing throughput of a master in a master-worker-style job.

The rest of the paper is organized as follows. Section 2 presents the usability of a network overlay for the middleware, and a P2P library supporting network overlay. In section 3, we propose an application of P2P-specific functions for the middleware P3. Section 4 and section 5 describe the design of the middleware, which reflects the proposed application. Section 6 shows the results of performance measurement, and the feasibility of the application method for distributed computation. In Section 7, we compare other middleware to P3.

2 A general-purpose P2P library supporting overlay

It is highly desirable to have a network overlay for P3 on which all computers can communicate equally and bidirectionally each others because the goals of P3 include mutual and equal transfer of computational resources. We made use of a general-purpose P2P library, JXTA [3], providing a network overlay on which any computer can initiate communication to other computers even though those computers are behind firewalls and NA(P)T. Such bidirectional and nonrestrictive communication also enables message-passing-style parallel processing, even though only a limited class of parallel processing, like master-worker, can usually be performed with firewalls.

On a network overlay provided by JXTA, a computer is called peer and it is distinguished from others by its peer ID. A communication target is specified using its peer ID. A peer can create and join multiple peer groups, in which a peer discover other peers, peer groups and communication pipes.

These functions provided by JXTA are primitives common to many kinds of P2P applications, such as file sharing, groupware, and instant messenger service. It requires a great deal of labor to accomplish distributed computation using the JXTA API directly. We developed middleware to facilitate management and development of parallel applications.

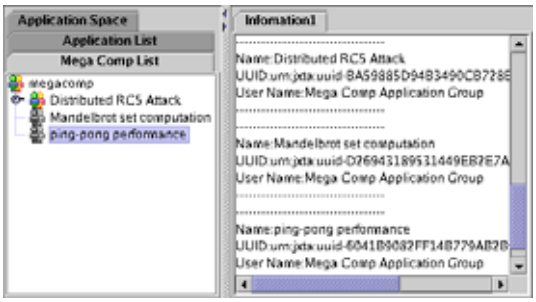

Figure 1: Job selection using the Host's graphical shell.

3 Application of P2P functions to P3

Besides network overlay, P2P-specific functions provided by JXTA can be naturally applied to, and efficiently support, distributed computation middleware, in addition to the usual P2P applications such as file sharing software.

Peer group: Parallel processing naturally involves collective communication to control multiple computers and broadcast a message to all computers. P3 creates a JXTA-supported peer group called a 'job group' for each job that is a submitted parallel application.

In a job group, in addition to collective communication by parallel applications, distribution of the parallel applications and data files and job execution control are performed. JXTA utilizes IP multicast and also relays a message across networks to achieve this broadcast.

Discovery: JXTA supports server-less decentralized discovery of resources, including peer, peer groups, communication pipes, and so on. It enables ad-hoc self-organization of computers.

In P3, a resource user creates a job group for his/her job and publishes an advertisement for the group. Resource providers discover the job group, determine whether they want to contribute to the job, and join the group if they do. Every computer is also discovered in the same way as a job.

4 Job Management Subsystem

P3 consists of a job management subsystem, a job monitor, and parallel programming libraries. This section describes the job management subsystem, and the next section depicts the libraries. The web-based job monitor shows the progress of a job and computers participating to the job group.

A P3 user manages a job using the following software.

- *Host:* The Host is a daemon program which a resource provider runs on his/her computer. It dis-

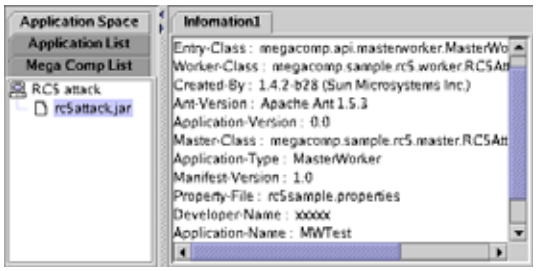

Figure 2: Job submission using the Controller's graphical shell.

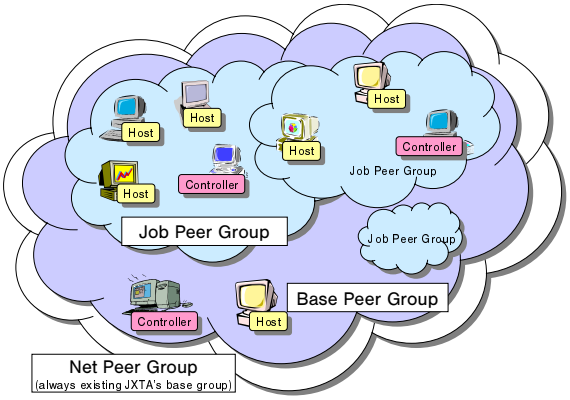

Figure 3: Organization of job management software and related peer groups.

covers a job group, receives a parallel application representing the job, and hosts the application.

- *Controller:* The Controller is a tool using which a resource user submits and controls jobs to a computer pool running Hosts (Figure 2).

The Host first verifies the digital signature of a discovered job. Secondly, if the Host is running in non-GUI mode, it decides whether to accept the job or not autonomically according to a policy supplied by the user. A policy has declarations representing acceptable jobs, for example, by the name of a job or a job submitter. The user running the Host can also make the decision by himself/herself on the GUI (Figure 1).

Figure 3 shows peer groups which Controllers and Hosts construct. The base peer group is the base group which all Controllers and Hosts create and join just after it is invoked. A user uses his/her Controller to use others' computing power by submitting a job, or runs a Host to contribute compute power to others. Figures 4 and 5 illustrate job submission using the Controller, and job participation by a Host.

A P3 application is written in Java language, and then compiled and packed into a JAR (Java Archive) file. A resource user submits those JAR files and data files, using a Controller. A parallel application can use

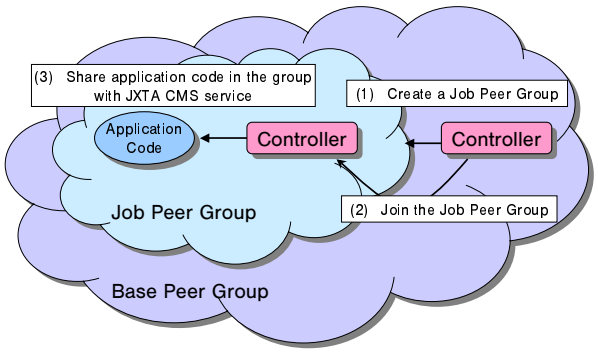

Figure 4: Controller's job submission process.

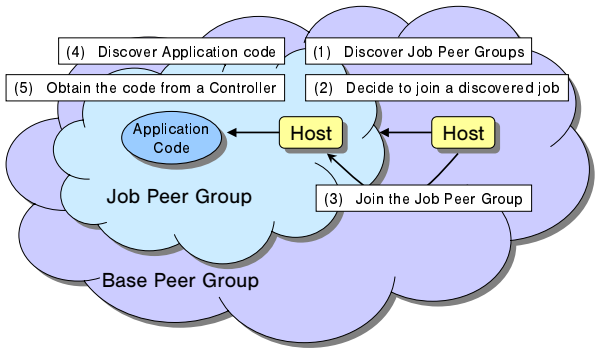

Figure 5: Host's job participation process.

parallel processing libraries (section 5) for inter-Host communication. A Host can also invoke a natively compiled application. But it will have better support later when P3 adopts a sandbox implementation, in which native applications run securely and a resource provider feels easy about accepting native applications.

5 Parallel Programming Library

P3 provides two parallel programming libraries for application developers. One of the common goals of such libraries is easy development of parallel applications. Developers should not have to deal with the details of communication and the underlying distributed environment. We designed the libraries along these certain rules. For example, one of the libraries, the master-worker library, can detect false computation results and an application on it does not have to do anything for the detection.

Figure 6 shows the structure of the parallel programming libraries of P3.

5.1 Object Passing Library

We provided an object passing library to support multiple parallel programming models with less development work. It is implemented directly on JXTA, hides the

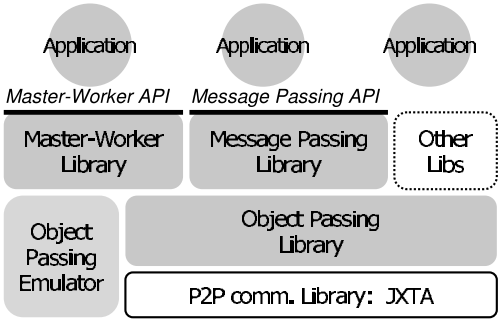

Figure 6: Libraries supporting parallel programming.

```
interface ObjectPassing {
  void send(PeerID receiver, Serializable obj);
  void broadcast(Serializable obj);
  Envelope recv(PeerID sender);
  Envelope recv(PeerID sender, long timeout);

  // receive a message from anyone
  Envelope recv();
  Envelope recv(long timeout);

  // check if a message is available
  boolean available();
  boolean available(PeerID sender);

  // register and unregister a message listener
  void addObjectPassingListener(
          ObjectPassingListener listener);
  void removeObjectPassingListener(
          ObjectPassingListener listener);
}
```

Figure 7: API of object passing library.

complexity of the JXTA API, and presents a simple set of APIs to libraries relying on it. The master-worker library and the message passing library could be easily implemented because they rely on the object passing library.

Figure 7 shows the API of the object passing library, by which an application can unicast or broadcast Java objects, and receive them synchronously or asynchronously. A communication target is specified using its peer ID. The raw JXTA API is sufficiently complicated to support all needs of P2P software. The object passing API wraps the JXTA API in a simple set of APIs to facilitate construction of various programming libraries.

5.2 Message Passing Library

The message passing library also sends and receives objects the same as the object passing library, but the communication target is distinguished using a non-negative integer rank, not a peer ID. This is the same as the method used in MPI [4]. This library maintains correspondence between a rank and a peer ID

```
class AMaster implements Master {
  Serializable getWorkerInitData() {
    return <data to initialize a worker>;
  }

  start() {
    while (...)  {
      WorkUnit wu = <a workunit>;
      // submit a workunit
      submitWorkUnit(wu);
    }
  }
}
```

Figure 8: Master side of an application program complying with the master-worker API.

```
class AWorker implements Worker {
  void init(Serializable initData) {
    // initialize this worker instance
  }

  WorkResult process(WorkUnit wu) {
    // process a workunit and
    // return the result
  }
}
```

Figure 9: Worker side of an application program complying with the master-worker API.

and translate one to the other.

The rank of a Host is assigned by a Controller when a job starts running, and a correspondence table is announced to all Hosts in the job group. Thus consistency of the message delivery target is assured because all Hosts share the table.

5.3 Master-Worker Library

The master-worker library supports master-worker-style parallel processing. An application developer writes programs for the master-side program, the worker-side program, and the workunit, respectively. A workunit represents a certain amount of work which collectively composes a job. It is delivered from a master to a worker and processed by the worker. Figures 8 and 9 show instances of a master-side program and a worker-side program.

Workunit delivery and scheduling are the charge of the master-worker library, not an application. A worker can join or leave a job at any time, even though the job has started running or the worker is processing a workunit. This feature, ad-hoc joining and leaving, is implemented entirely by the library, and an application developer does not need to take care of it. If a worker leaves a job group, a workunit delivered to the worker but not completed is delivered to another worker, and the whole of a job is accomplished after all.

We have to assume that there are malicious participants in Internet-wide distributed computing. They may receive workunits and not process them. It is also possible for them to return false calculation results to obstruct the job.

The former lazy worker problem is addressed by the timeout-based redistribution of workunits. The latter false results problem, is also dealt with by the master-worker library automatically. The library detects a false result by *voting* [5]. In P3, if a false result is detected, the workunit is distributed again. This detection process is performed completely by the library and an application does not deal with false results.

A master distributes a single workunit m times to workers and compares n returned calculation results. The master accepts a result as the correct one if n results agree. An application developer can provide code performing a custom matching process in which n results are examined, while the default code in P3 performs exact matching. Such a custom matching process allows automatic verification to work for results like floating-point values, which have some play in them and cannot be compared exactly.

A Host, not Controller, takes the role of master. In P3, a Controller does only job management work. A Host taking the master's role is chosen by the Controller submitting the job. It is possible that a Host stands as a candidate for master. A user can specify it when invoking a Host. If a Controller finds multiple standing Hosts, it chooses one standing Host randomly. If there is no standing Host, a Host is chosen randomly out of all Hosts in the job group. The Controller announces the chosen Host as the master and all Hosts recognize it. The current master selection process is naive, but it can be improved on future work involving the capabilities of Hosts, like computing power and network bandwidth.

Consequently, one of the Hosts participating in a master-worker-style job performs the work of a master.

5.4 Emulator

It is time-consuming work to develop and debug a parallel program, compared to a sequential program. The reasons for this include difficulty in understanding parallel activities in the program in addition to longer turnaround time to set up and run the program.

P3 includes an emulator of the object passing library. The emulator hosts and runs a parallel application on a single computer (Figure 6). The emulator supports both master-worker-style applications and message passing applications because both libraries rely on and use only the object passing API for communication. This generality of the emulator is derived from the generality of the object passing API and the layered design of P3 libraries.

Table 1: Communication latency.

TCP (C language)	0.062
TCP (Java language)	0.064
Message passing library	4.5
	(msec)

6 Performance Evaluation

This middleware benefits from a rich set of P2P-specific functions provided by a P2P library as described in section 2 and 3. It is naturally expected that such rich functions introduce a certain amount of overhead into communication performance and application throughput. We measured both to evaluate the feasibility of the P2P library, JXTA.

All experiments were carried out on a PC cluster which consists of 32 computers connected over 1000BASE-T Gigabit Ethernet. Each computer has dual 2.4 GHz Intel Xeon processors and runs Linux 2.4.19. The Java runtime is HotSpot Server VM of Java 2 SDK 1.4.2. The JXTA implementation is version 2.1 of the J2SE reference implementation. We ensured that JXTA uses TCP as the communication protocol by prohibiting use of HTTP.

Bandwidth on the Gigabit Ethernet LAN is much broader, and latency on it is smaller than that of the current Internet. But such a rich environment can bring performance, limitations, and overhead of the middleware itself, to light.

6.1 Communication Latency and Throughput

We measured the communication latency and throughput of the message passing library of P3 and compared the results with the latency and throughput of raw TCP. The comparisons illustrate the overhead introduced by JXTA and the message passing library.

Table 1 shows one-way latency which was measured by 1000 round trips of a 1 byte message. The JXTA-based message passing library took about 4 or 5 msec processing time on computers involved in the communication. The latency was introduced mainly by JXTA and the parallel programming libraries on it, not from the network itself, because the underlying TCP communication costs only 0.06 msec.

Throughput shown in Figure 10 is measured by 100 round trips of a variously sized message between 2 computers. In this paper, kilo (K) means 1024, not 1000, mega (M) represents 1024^2 and so on. The highest

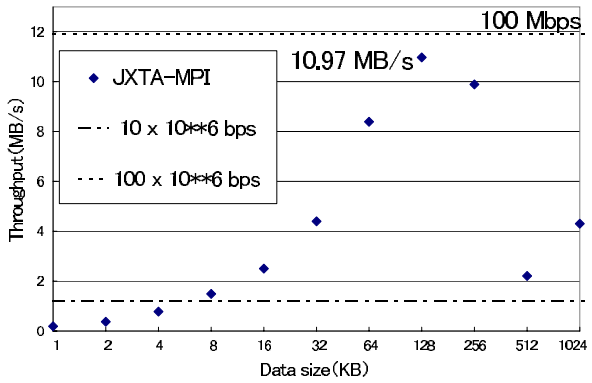

Figure 10: Throughput of the message passing library on JXTA.

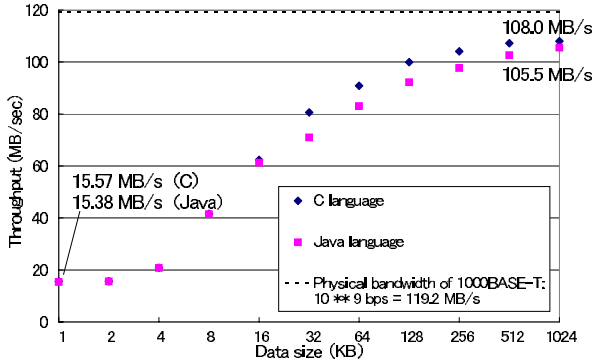

Figure 11: TCP throughput.

throughput near 100 Mbps was achieved with 128 KB sized messages. The same throughput can be achieved if a message is larger than 128 KB because such a message can be divided into smaller chunks.

Raw TCP throughput reaches about 90% of the physical bandwidth of Gigabit Ethernet as shown in Figure 11. Compared to this, the JXTA-based message passing library could achieve only about 9% of the physical bandwidth. It is natural to conclude that such low throughput is due to JXTA because the library is a thin layer which just packs a Java object into a JXTA message and send it. A report [6] shows that JXTA 2.2.1 achieves 136.78 MB/s on a high-speed network Myrinet. This result indicates that JXTA can potentially fulfill the bandwidth of Gigabit Ethernet with appropriate settings of underlying network layers. Anyway, Figure 10 demonstrates that the JXTA-based message passing library could fulfill the bandwidth of today's Internet connection to homes and small offices, which is currently up to about 100×10^6 Mbps.

6.2 Throughput of Workunit Passing

In master-worker-style parallel processing, a master is prone to be the bottleneck of the whole parallel computation as the number of workers increases. It is important to prevent this for efficient parallel processing. The work of a master consists of preparation of workunits, distribution of them, collection of processed results for each workunit, and post-processing of the results. Two processes, workunit distribution and result collection, are performed by the master-worker library, not an application. Performance of the two processes can be limiting factors of the entire parallel processing effort. They should be as efficient as possible.

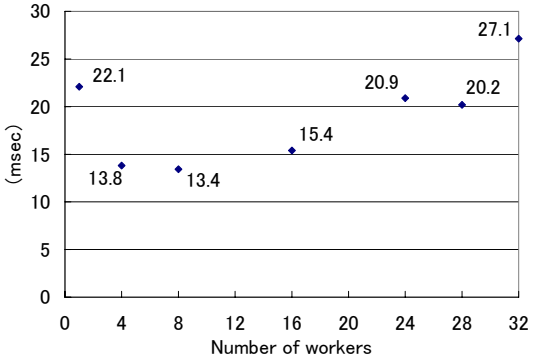

Figure 12: Workunit handling time.

We measured the throughput represented by the number of workunits (results) a master could distribute and collect in a unit of time. The time for a pair of distributing and collecting operations was measured by distributing 200 workunits and then collecting calculated results for them. Workers do not perform any work on a workunit and just return a result.

Figure 12 shows the results. A pair consisting of a distribution and a collection took 28 msec in the worst case and 13 msec in the best case. In other words, 35 workunits were processed a second in the worst case and 76 workunits a second in the best case. This number would be improved by increasing workers if the latency was caused by the network, but unfortunately this is not the case because the cause is message processing time, not the network (section 6.1).

In an Internet-wide project operated by United Devices, the throughput has been adjusted to 120 workunits a second by adjustment of the size of the workunit [7]. The project involves tens of thousands of computers. It is natural to expect that the peak performance of that system is several times as high as the average throughput of 120 workunits per second. It was found that JXTA and the master-worker library based on it have to be improved several times if it hosts tens of thousands computers with a single master. A P3 system with many computers have respective masters for each job, though.

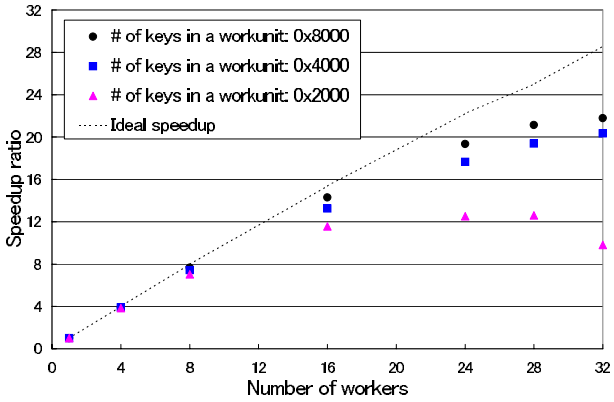

Figure 13: Parallel efficiency in an RC5 key search according to the granularity of workunits.

Table 2: Calculation time per workunit.

Number of keys in a workunit	Processing time (sec)
0x8000	1.4
0x4000	0.69
0x2000	0.36

6.3 Tolerance to Fine Grain Workunits

In a master-worker-style application, it becomes easy to gain performance improvement as the granularity gets coarse because communication overhead is then hidden by calculation time. We measured the speedup according the number of workers. The purpose of this experiment is estimation of tolerance to fine-grain workunits.

The application used in this experiment is a brute-force key search on the RC5 cryptosystem, which is the same computation as that of Project RC5 of distributed.net [8]. It was implemented on P3's master-worker library. The granularity of a workunit is adjusted using the number of key candidates in a workunit, which is set at 0x8000, 0x4000 and 0x2000 (hexadecimal notation). The number of workunits is 200 and the number of workers is varied from 1 to 32. The performance is represented by the number of workunits processed in a unit of time.

Figure 13 shows the speedup ratios according to the granularity of workunits. The ideal speedup ratio is not the same as the number of workers because the number of workunits is not a multiple of the number of workers. Table 2 shows the calculation time of a workunit. Load imbalance is the cause of the limited speedup ratios with 0x2000-sized workunits. In the case of 0x2000 and 24 workers, the total processing time was only 6.13 sec, and it takes about 1 sec for a worker to start running from the time a master instructs the workers to start. This varying delay caused the load imbalance and limited speedup.

Speedup ratios shown in Figure 13 indicate that the JXTA-based master-worker library can achieve about a speedup of 20 fold with 32 workers even though a workunit is adequately fine grained as it completes in a second. Note that such fine and severe granularity is not realistic in an Internet-wide project. The granularity is usually adjusted to several or dozens of minutes.

7 Related Work

P3 has the following features because its goal is mutual and equal transfer of computational resources.

- Anyone can use others' resources by submitting a job.
- A resource (PC) provider can choose jobs to which his/her computer contributes.

P3 is compared with other distributed computing middleware in this two respects in Table 3. At least, job submission requires an application to be independent of the middleware. Table 3 shows application independence of middleware in place of the job submission feature. Job selection, the latter feature, means that a calculating software (i.e. the Host of P3) can accept multiple jobs simultaneously and can also reject a job.

JNGI [9] is a distributed computing framework. It is also based on JXTA as P3 is. JNGI manages a large number of peers efficiently by dividing peers into peer groups which have a limited number of peers each, while P3 utilizes a peer group to wrap activities of an application into it. A JNGI paper [9] could not evaluate JXTA's feasibility for distributed computation because the paper does not show the standard of speedup ratios obtained even though the experiment is performed with heterogeneous computers. And the experiment did not impose much of a load on JXTA. 150 workers were involved in the experiment but the granularity of a workunit is as coarse as 130 sec. The master deals with only about one (150/130) workunit a second in that situation.

BOINC [10] is middleware that helps creation and operation of public-resource computing projects. BOINC is application-independent and provides a false results detecting mechanism as P3 does. A computing client of BOINC can accept multiple projects (jobs) simultaneously as P3 can, though a project originator has to prepare a computer to host the project.

A number of Internet-wide distributed computing middleware [11, 12, 13, 14] have been developed, and they each have respective characteristics, like product-level quality and special programming models. P3 can

Table 3: Comparison between P3 and other Internet-wide distributed computing software.

	Application-independent middleware	Job selection by resource provider	Communication protocol	Programming Languages	Parallel programming model	Detection of false results
P3	**yes**	**yes**	JXTA	Java	Master-worker, Message passing	**yes**
JNGI [9]	yes	no	JXTA	Java	Master-worker	no
BOINC [10]	yes	yes	HTTP and original	C, C++, etc.	File	yes
XtremWeb [11]	yes	no	choosable (RMI, etc.)	C, C++, etc.	File	
SETI@home [1]	no	no	HTTP	C, C++, etc.	File	yes

be differentiated by the job selection feature, automatic false result detection, and support of message passing programming.

8 Conclusion

We have presented middleware for mutual and equal transfer of computing power between individuals. Another and traditional goal of the middleware is large-scale distributed computing. We applied P2P functions, such as discovery and peer groups, provided by JXTA, a general-purpose P2P library, to the middleware. Utilizing these functions, computers can construct a group ad-hoc, and the group can process a submitted parallel job.

Performance measurement showed that P3 can fill 100 Mbps bandwidth. Workunit handling time came up to about 30 msec in the worst case and it should be improved to host tens of thousands of computers in a job. A master-worker-style job could achieve a 20 fold speedup with 32 workers even with fine grain workunits.

Future work includes confirmation and improvement of quality and feasibility for real world use, while real applications, including protein folding [15], have already been implemented. It would also be interesting to integrate P3 with Grid middleware which harnesses multiple PC clusters. In that case, P3 would take charge of cluster management software like PBS or LSF.

Acknowledgments

This work was partly supported by the Information-technology Promotion Agency (IPA) "Next Generation Software Development" project.

References

[1] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, "SETI@home: Massively distributed computing for SETI," in *Computing in Science and Engineering*, vol. 3, pp. 78–83, January/February 2001.

[2] distributed.net, "distributed.net: Node zero." http://www.distributed.net/.

[3] Project JXTA, "jxta.org." http://www.jxta.org/.

[4] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[5] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," *Future Generation Computer Systems (FGCS)*, vol. 18, pp. 561–572, Mar. 2002.

[6] M. Jan and D. A. Noblet, "Performance evaluation of JXTA communication layers," Tech. Rep. RR-5350, INRIA, Oct. 2004. http://www.inria.fr/rrrt/rr-5350.html.

[7] K. Ueno, "Personal communication," June 2003.

[8] distributed.net, "Project RC5." http://www.distributed.net/rc5/.

[9] J. Verbeke, N. Nadgir, G. Ruetsch, and I. Sharapov, "Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment," in *Proc. of Third International Workshop on Grid Computing (GRID 2002)*, pp. 1–12, Nov. 2002.

[10] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proc. Fifth International Workshop of Grid Computing (GRID 2004)*, pp. 4–10, Nov. 2004.

[11] G. Fedak, C. Germain, V. Néri, and F. Cappello, "XtremWeb: A generic global computing system," in *CC-Grid2001 Special Session Global Computing on Personal Devices*, May 2001.

[12] GreenTea Technologies, Inc., "GreenTea Platform Whitepaper," 2002.

[13] M. O. Neary, A. Phipps, S. Richman, and P. Cappello, "Javelin 2.0: Java-based parallel computing on the internet," in *Lecture Notes in Computer Science (LNCS) for 6th Int'l Euro-Par Conference (Euro-Par 2000)*, vol. 1900, pp. 1231–1238, Springer Verlag, Aug. 2000.

[14] L. F. G. Sarmenta and S. Hirano, "Bayanihan: Building and studying web-based volunteer computing systems using Java," *Future Generation Computer Systems (FGCS)*, vol. 15, pp. 675–686, Oct. 1999.

[15] I. Ono, H. Fujiki, M. Ootsuka, N. Nakashima, N. Ono, and S. Tate, "Global optimization of protein 3-dimensional structures in NMR by a genetic algorithm," in *Proc. of 2002 Congress on Evolutionary Computation*, pp. 303–308, 2002.