

P2P-Tuple: Towards a Robust Volunteer Computing Platform

Lei Ni and Aaron Harwood
NICTA Victoria Research Labs, Australia,
Department of Computer Science and Software Engineering,
The University of Melbourne, Australia
<http://p2p.csse.unimelb.edu.au>

Abstract

The evolution of emerging Volunteer Computing systems is placing an ever greater emphasis on the use of a loosely coupled decentralized architecture, mainly to address concerns of scalability and robustness. We propose to build the next generation Volunteer Computing systems on top of well studied Peer-to-Peer techniques to fully take advantage of its decentralized characteristic and its very large, shared data storage capacity. This paper sketches the design of such a completely decentralized and fault tolerant Volunteer Computing system by using a Peer-to-Peer based Tuple Space formed by volunteer peers. Both simulation and software prototype evaluation results show support for the proposed design even when all peers are having a high churn rate of Mean Time Before Failure of 2 hours.

1 Introduction

Volunteer Computing [2] (VC) platforms like BOINC [1] are the result of a decade long ongoing architectural evolution started since the late 1990s. The current generation architectures, for which BOINC [1] is a widely known example, are based on the client-server architecture. In such VC systems, a *task* is typically defined as containing anywhere from dozens to millions of independent *jobs*, they are submitted by the user or automatically generated by a Job Generator. A *central* server is usually used to assign these jobs to voluntarily contributed machines, synonymously known as *volunteers*, *workers* or *executors*. Once finished, the output of each job, called a *result*, is pushed back to the originating central server for validation and storage. Observe that for this straight forward approach: (i) the overall system's availability depends on the availability of the central server, and (ii) it is difficult to share storage capacities between workers as they usually operate obliviously to each other. Neither is it technically feasible to require the server to coordinate the exchange of data between clients due to the significantly small network bandwidth and processing power of the

central server relative to the aggregate bandwidth and processing power of the workers. As an example of the above issues in real deployment, we have developed a BOINC based distributed network measurement system, during the first three months of 2009 we had two system failure incidents that stopped the operation of the system for 2 days and both of them are related to central server failure.

1.1 Contribution

The main contribution of this paper is the proposed architecture of a P2P based VC platform called P2P-Tuple which removed all central components common in previous VC designs thus achieves high robustness. The P2P based design allows all distributed nodes in P2P-Tuple to disconnect or crash at anytime but that will not fail the submitted task. Our experimental results show that P2P-Tuple can handle very high peer churn [13] rate and still provide competitive speedup.

2 Related Work

CCOF [8] is a P2P based VC platform in which member peers construct a CAN [12] Distributed Hash Table (DHT) overlay. A central scheduler, called the Application Scheduler, aggregates peers from the overlay and schedules jobs onto them. The returned results will be sent to this Application Scheduler for validation and storage. By exposing itself to all untrusted peers during the full life cycle of jobs and making all scheduling decisions, the Application Scheduler shares the same properties as BOINC's central server.

Harvard's TONIC project [10] is a pioneer for exploring Tuple Space's [6] application in scientific parallel processing. In TONIC, a central storage server is used to host the Tuple Space for storing jobs and results, a lookup service is also provided on that server to allow discovery and accessing of the Tuple Space. Such design successfully separated the job generation and processing to allow jobs to be processed while the workers do not have to know anything about job generation. The centralized Tuple Space and its lookup service which are hosted on one or more central

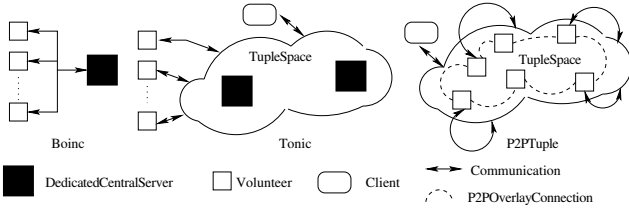


Figure 1. BOINC, TONIC and P2P-Tuple’s architectures compared.

servers is a single point failure and bottleneck to scalability, workers in TONIC still can not contribute and share storage capacities.

3 Design of the P2P-Tuple

3.1 High-level Architecture

A high-level architectural comparison of BOINC, TONIC and our proposed P2P-Tuple system is depicted in Fig. 1. P2P-Tuple: (i) is completely decentralized, peers form a DHT overlay on which the Tuple Space [6] is constructed by using the storage capacities contributed by peers. Peers pull jobs from the Tuple Space and push results back to be stored on the Tuple Space so they can be collected by the job owners later or be used to feed the next stage computation in a work flow configuration; (ii) existing systems like BOINC and TONIC assumes executors to be unreliable while the central server or scheduler helps to ensure the eventual completion of all jobs. In P2P-Tuple, we only assume the P2P based distributed storage system known as the Tuple Space can be made of high reliability with appropriate fault tolerance designs. Any machine in the system can fail/disconnect at any time, the overall system will still be functional and complete submitted jobs.

3.2 Overall operation

In P2P-Tuple, a single Pastry [15] DHT overlay is formed in which each volunteer node becomes a Pastry peer. Peers can choose to join one of the SCRIBE [3] based *groups* which are equivalent to the “project” concept in BOINC, each such group is owned by a *group owner* (e.g. a researcher). Groups advertise themselves on the Internet through their web sites and they are identified by the URLs of the web sites when volunteers choose which group to join. Joining any group would form a trust relationship between the volunteer and the group owner, allowing programs signed by the group owner to run on the volunteer machines.

The objective of our design is to support the following overall operations of the system during task’s life-cycle:

- Group owners submit their tasks using a client program – the client connects to any peer in the network and the task’s related data is uploaded to the Tuple Space which includes transparent replication and/or

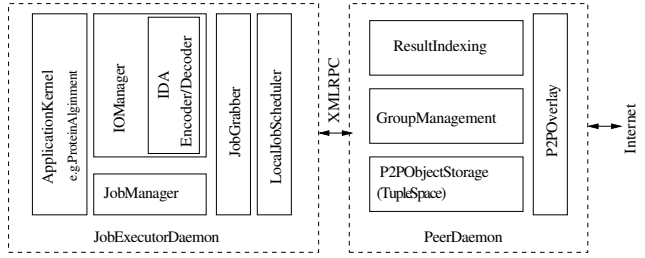


Figure 2. The single peer architecture.

encoding with an IDA [11] erasure code algorithm for data reliability purpose.

- The client periodically broadcasts the *task notification* message informing all peers in its group of the recent submission of the new task. On receiving this notification, which contains a *task_id*, peers in the group start pulling jobs associated with that task for processing; using a randomized pull strategy. Each submitted job can be addressed by the combination $\langle task_id, job_id \rangle$ in the Tuple Space where *job_id* is a numeric identifier of the job.
- Peers notified by the above broadcast will also broadcast the received task notification to other group members every few minutes, called the *broadcast interval*, unless the task is terminated or the peer recently received such a task notification during the previous broadcast interval. This allows peers, that come online after the task submission, to be notified and to begin work on the task.
- Once any job is finished, an IDA encoded result is pushed back onto the Tuple Space. A distributed *job status vector* (bit vector) is used to record the completion of each job and thereby the availability of its result.
- A task is terminated and all peers in the group become idle again when the client concludes all jobs are completed, as indicated by the job status vector, or otherwise when the task has exceeded its maximum allowed runtime which is a parameter specified during task submission.

3.3 Single Peer Design of the P2P-Tuple

On the peer level, as shown in Fig. 2, each peer provides two top level services: Tuple Space storage implemented by the Peer Daemon and job execution implemented in the Job Executor Daemon. Each peer can thus contribute storage space by only running the Peer Daemon, or can contribute both storage and CPU cycles by running both daemons on their local machine.

The **P2P Object Storage Service** uses PAST [16] to form the Tuple Space. Data stored in this service is associated with a key and expiration date, large files are di-

vided into 64Kbytes chunks and encoded using the IDA algorithm [11]. A simple $Put(key, type, data)$ and $Get(key)$ interface is provided to access the Tuple Space on the object level. The *in* operation which reads and removes the content in the traditional Tuple Space design [6] for exclusive access is not included due to the fact that the peer holding the content could disconnect and cause permanent loss of that data.

The **Group Management Service** uses SCRIBE [3] to maintain tree based multicast groups on top of the P2P overlay. Task submission and termination notifications can be broadcasted to all group members with small number of messages per peer cost. We also implemented a random walk on this tree to allow randomly picking one or more peers from the group for exceptional situations. For instance, when a job is finished and the result is uploaded, it is still possible that the IDA encoded result can become permanently lost (e.g. due to the churn) before it is collected by the client (described in Section 3.4) or before being taken as input by the next stage computation in a workflow (described in Section 3.5). We handle such failure by letting the consumer of the result, such as the client or the peer in work flow’s next stage computation, to randomly pick peers and push the job associated with the damaged result to those peers to re-run the job.

The **Result Indexing Service** maintains status of all submitted jobs across the P2P network. For each task, buckets of 128 continuous jobs’ status, as 128 bit *job status vectors*, also known as *slices*, are indexed onto peers. Each bit of the slice represents whether the corresponding job has been completed or not. Each slice representing 128 jobs is replicated 16 times. On job completion, status change messages are sent to all replicas to update the content of the corresponding bit. This service also periodically synchronizes between all replicas which may involve recreating lost replicas to maintain a constant replica count. We assume that false positives are non-existent (peers never falsely report a job to be done) and therefore to query the status of a job, parallel lookup operations are issued to download all replicas which are subsequently combined using the bit-wise OR operation. This further serves to eliminate any false negatives that arise from inconsistent state information. Peers thereby determine which of the 128 jobs in the slice haven’t been completed.

When working on a task, each peer uses the local **Job Grabber** to independently pull and cache a number, L_q , of unprocessed, randomly chosen jobs and their related data, including the description of the job and input data. This main purpose is to allow a controlled level of job duplication – once the job enters the queue then it will be processed, even if some other peer has already processed that job in the meantime. This time-of-check-to-time-of-use mismatch is intentionally added so that peers can control the percentage of jobs that would be duplicated (i.e. independently run

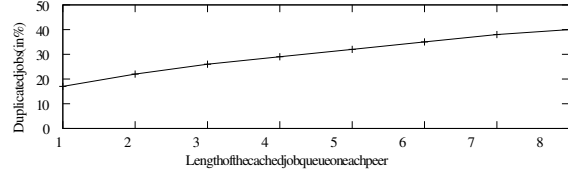


Figure 3. The percentage of duplicated jobs.

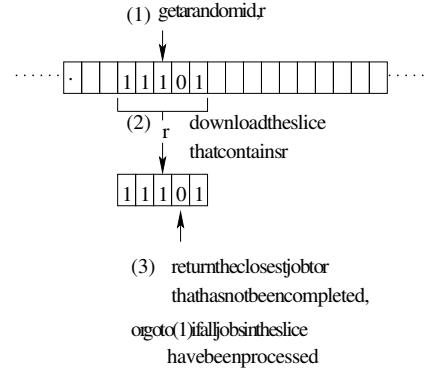


Figure 4. The job scheduling policy on each peer.

by multiple peers). Fig. 3 shows the relationship between the value of L_q and percentage of Job duplication, obtained experimentally using simulation.

The **Local Job Scheduler** implements an efficient random scheduling policy where each peer randomly pick jobs that have not been completed (by querying the above Result Indexing Service), skipping over completed jobs. Similar random scheduling policy is also used in [9].

Fig. 4 shows the procedure that the local scheduler applies. The scheduler picks a job id uniformly at random within the range of jobs and downloads the slice that contains that job id. It then scans through the slice starting at the position of the job id and moving to the right (with wrap around to the beginning of the slice), looking for the first available uncompleted job. If all jobs in the slice have been completed then it picks a new random job id and repeats the procedure. In the workflow case, the local scheduler will pick the first job that has the closest job id to the random id that has all dependent input data available on Tuple Space; more details are given in Section 3.5. The local scheduler is modularized and can be easily replaced to use other policies/algorithms.

The **IO Manager** applies replication or the IDA [11] erasure code on all I/O data to be stored on the P2P network, depending on data size. The IO Manager provides two methods, download and upload, for accessing the Tuple Space at the file level. The **Job Manager** prepares the running environment for the parallel application process (called **App Kernel**) monitors the life cycle of the process, then collects result on its completion.

Table 1. Member methods in a client.

createTask(exec_name, size)	create a task
createJob(task_id, job_id)	create a job description

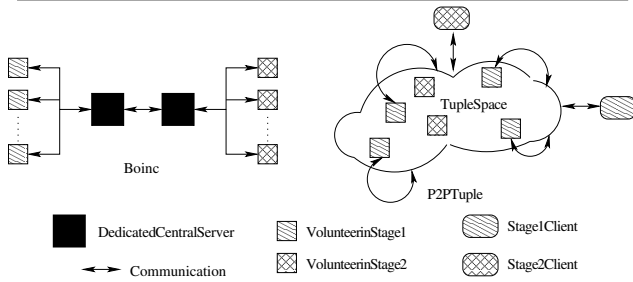


Figure 5. BOINC and P2P-Tuple's workflow configurations compared.

3.4 Application

To port existing applications which are suitable to be parallelized as master-worker programs to the P2P-Tuple platform, a C++ class with two member functions shown in Table (1) need to be implemented. This client program is used to generate task, divide it into multiple independent jobs, upload all of them and collect results back from the Tuple Space as/when specified by the user. The client only creates tasks, defines all jobs and uploads the serialized task related data to the Tuple Space – no scheduling or validation is done by the client. For tasks that may take hours or even days to finish, the client can disconnect at anytime, process checkpointing allows it to restart from the previous known status. Another capability of the client is job pushing. The client may push jobs to peers, to force their execution. It will typically do this for the remaining small percentage of unfinished jobs for faster task completion. Currently, if the client is online, it will push the last 2% of jobs. This helps to eliminate tail effects where pure randomized pulling becomes inefficient. Please note, tasks can eventually complete without such push operations and the client doesn't have to always be online, such pushes are only optional and serve to speed up the completion of the remaining few jobs.

3.5 Robust Workflow Support

Considering two tasks, as both task's input and output data are stored in the Tuple Space, it is possible to specify the output of a task as the input of the next task to form a simple pipeline style workflow in the P2P-Tuple. The high-level architectural comparison is shown in Fig. 5, indicating that P2P-Tuple avoids a centralized pipe for results communication between stages. In BOINC's case, a failed server in any stage can block the whole pipeline. P2P-Tuple clearly avoids such bottlenecks by maintaining all job data and intermediate results in the distributed Tuple Space, which promises both high combined bandwidth and availability.

When working on pipeline style workflow tasks, details of input data are a part of the job description stored and

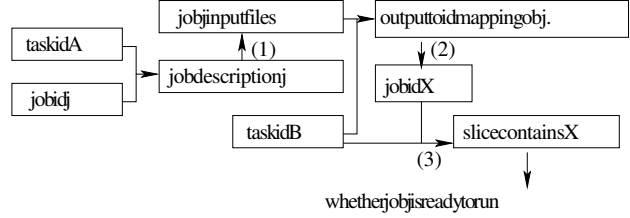


Figure 6. Data dependency in workflow.

indexed in the Tuple Space. For each job we also maintain the mapping information from the output file details to the *job_id* that generates such files. All above data forms the dependency graph between jobs and a working example is shown in Fig. 6. In this example, when a job with id *j* in task A is picked randomly by the local scheduler, the system: (i) finds job *j*'s input files from the job description of job *j*, (ii) finds all jobs in task B that generate outputs that map to job *j*'s input files, job *X* in this example, and (iii) downloads all associated slices for those jobs in task B, to check whether they have been completed and thus whether all of the input for job *j* is available. The local scheduler will loop to the beginning and randomly choose another job id of task A if not all dependent jobs in task B have been completed.

4 Fault Tolerance

4.1 Erasure Code and Replication

The robustness of the overall P2P-Tuple system heavily relies on the availability of all stored jobs and results in the Tuple Space. As a good example on the challenging environment facing VC systems, the Entropia project reports the average Mean Time Before Failure (MTBF) of Entropia nodes is only 2-3 hours during weekdays and 6 hours during weekends [7]. From now on, we present churn rate in MTBF, higher peer MTBF means lower churn rate, for better readability.

Erasure codes such as Rabin's IDA algorithm [11] allow the encoding of S bytes of data into K_{ida} chunks each with $\frac{S}{M}$ bytes ($K_{ida} > M$, $M > 1$); where reconstruction of the original data can be done with any M such chunks. The parameters K_{ida} and M thus can be tuned to suit different data reliability requirements and the ratio of $R = \frac{K_{ida}}{M}$ indicates the redundancy level; where R times the storage space and communication is required to store all K_{ida} encoded chunks. On the other hand, in replication, the same data is uploaded K_r times to be stored at different locations. The value of K_r , usually known as the number of replicas, is the redundancy level. Previous studies have already concluded that the benefits of erasure code are that higher data reliability can be achieved when incurring the same redundancy level as full replication.

However, erasure code does add significant latency overhead when at least M IDA chunks must be downloaded to reconstruct the original data. When downloading small data in which access time is dominated by communication la-

tency, erasure code can produce M times more latency. We use IDA erasure code for input and output data, while job descriptions, task description and other data items that have expected small sizes are only replicated.

4.2 Repairing Strategy

Whether lost objects should be regenerated, known as repairing, has been discussed by the community [14], we have found that repairing is essential in P2P-Tuple. When the MTBF of volunteers that form the Tuple Space is only hours then most stored chunks/replicas will be lost very quickly, e.g. if we assume that the peers' MTBF is close to Entropia nodes of 2-3 hours, and such departures are of exponential distribution, as suggested by [17], then according to the exponential distribution's Cumulative Distribution Function, around 86% of the stored replicas or IDA chunks will be lost after 6 hours. It is still possible to access some of the stored data when the redundancy level is high, but all access requests will have to be handled by that 14% of survived volunteers, which in turn is a scalability issue because those long surviving nodes will have to handle the increasing number of accessing requests without such object repairing.

In our Tuple Space, each stored object has an identical "mirrored replica" and they form a pair that will periodically report to each other their existence. Detected data loss will be restored and a stored object is only permanently lost when both peers that hold the objects in the pair disconnect within a short period of time. To also counter the high bandwidth consumption for maintenance, we remove stored data as soon as they are no longer required.

4.3 Job Duplication

Using the randomized job scheduling scheme, some jobs will be duplicated on multiple peers, this is for three reasons. First, duplicated jobs can increase the probability of job completion when peers keep leaving throughout the duration of the task. Secondly, the results of the duplicated jobs provides extra redundancy. When the results of the duplicated jobs are uploaded onto the Tuple Space, the lost IDA chunks or replicas of the result are re-uploaded and thus the availability will be reset to 100%. The third reason is Volunteer Computing uses untrusted resources and thus it needs validation to maintain an effective level of trust for the results. Two separately generated results for the same job can be compared and once conflict is detected between these two results, the same job can be pushed to more peers and the results are used to form a vote of confidence. Suspicious peers can thus be detected and excluded from the DHT.

4.4 Checkpoint

The progress of the client, including how many jobs have been generated and uploaded, how many results have been collected back from the P2P network and etc., are periodically

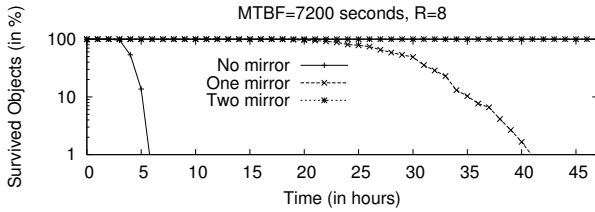
stored into a checkpoint file on the local file system to allow restarting the client process when necessary. The size of the checkpoint image is determined by the total number of jobs in that task. An uncompressed checkpoint image for a task of 1 million jobs is only about 10Mbytes. Such compact checkpoint image size allows us to store the checkpoint image in the Tuple Space which provides support for migrating the client between workstations.

5 Evaluation

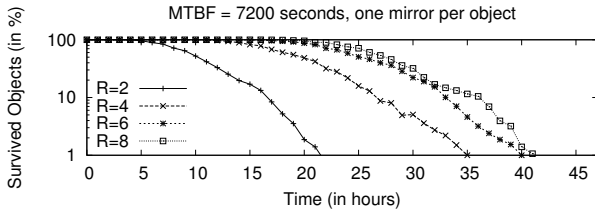
The main hypothesis of P2P-Tuple is that all job related data stored in the Tuple Space can be of high availability. We verify this hypothesis using simulation.

The FreePastry software used to build the Tuple Space supports the production code to be simulated in a time compressed event driven simulation mode in which hundreds of peers can run on the same machine with all network traffic short circuited. A 300-peer network is simulated on a 2.6GHz Intel quad core PC, using the same Tuple Space code base with minimal modification. During the simulation, hundreds of IDA encoded files are uploaded to the Tuple Space first, the M parameter is set to be 8 which is the default value in P2P-Tuple, "mirrors" are checked every 10 minutes and lost ones will be repaired when possible. We then measure the percentage of how many of these files can survive over time with different numbers of mirrors and redundancy level R under different churn rates. Each peer is given a life time that is of the exponential distribution with the specified rate when joining the network and it will be killed with all hosted data discarded at the end of its life time. The killed peer will re-join the network as a fresh peer shortly after its death to maintain a stable network size. Depicted in Fig. 7(a), according to the simulation results, when $R = 8$ and there is no mirror, the availability of all stored data quickly downgrades to close to zero in about 6 hours, while with the same R and only one mirror, 99.99% data survival rate can be achieved for the first 16 hours. This shows that those mirrors are essential in a normal churn rate environment. Increased survival rate is possible when two mirrors are employed with the cost of extra communication overhead on detecting lost copies of mirrors and repairing them. Actually at least 48 hours of 99.99% data survive rate is observed in our simulation when $R = 4$. The impact of R is shown in Fig. 7(b); basically the total hours of 99.99% data survive rate is doubled when R doubles. Data survival rate in a high churn environment (MTBF=1800 seconds) is visualized in Fig. 7(c), in such a harsh environment, high object survival rate can still be sustained for a couple of hours, with both two mirrors and high redundancy level. Based on the above simulation evaluation, we use one mirror and $R = 8$ for the rest of the experiments and such configuration allows submitted jobs and results to be cached on the Tuple Space for up to 16 hours with at least 99.99% availability. Small data replicated and stored in the Tuple

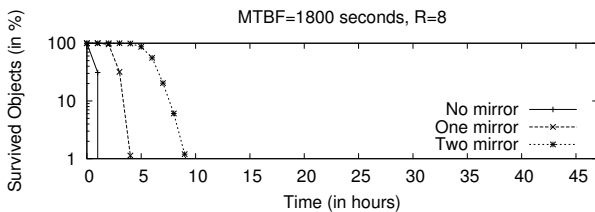
Space can be set to have large K_r value, e.g. $K_r = 2R$, to maintain high availability.



(a) MTBF=7200 seconds, R=8



(b) MTBF=7200 seconds, one mirror per object



(c) MTBF=1800 seconds, R=8

Figure 7. File survive rate over time using different fault tolerance parameters under churn.

Table 2. Per job I/O sizes and typical applications.

	Size	Typical Application
Class A	10Kbytes	Small protein in Folding@Home
Class B	80Kbytes	between Class A and C
Class C	320Kbytes	SETI@Home

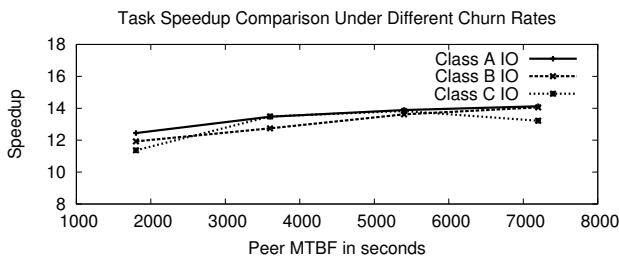


Figure 8. Speedup under different churn rate.

The speedup, bandwidth and fault tolerance performance of the P2P-Tuple prototype is first studied on a 20 node cluster using a “dummy application”. This dummy application

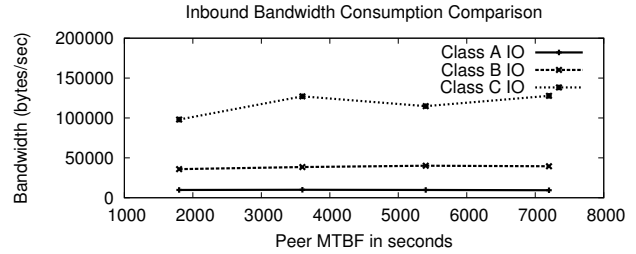


Figure 9. Bandwidth consumption.

allows users to set runtime, input/output data size per job as parameters for synthetic tests of different applications. The per job data IO sizes used in this section is detailed in Table (2); the default per job runtime is set to be one minute unless otherwise mentioned. A churn generator is implemented to create churn events, when any peer joins the system, the churn generator will assign a *life time* l_t , randomly picked according to the exponential distribution of the specified MTBF value, and that peer will be stopped using the UNIX KILL signal after l_t seconds. The killed peer will be restarted as a new peer after 30 seconds waiting period with a different random peer id, all previously stored data and associated job process will be discarded after restarting. All experiments below done in such faulty environment shows the good fault tolerance of the P2P-Tuple.

First we run one P2P-Tuple instance on each of the 19 nodes (client runs on another node) and submit tasks each having 512 independent jobs. We set the job queue length to be 1 to minimize the duplicated job runs and there is no results validation. The average runtime performance measured in terms of speedup under different churn rates is shown in Fig. 8. Churn causes the same task to take longer to complete on high churn rate (i.e. low MTBF). The speedup is about 14 using 19 machines because: (i) churn would cause the computation time spent on unfinished jobs to be wasted when peers quit the network, and (ii) the random job scheduling produces about 17% runtime overhead when the job queue length is 1, as shown in Fig. 3.

The bandwidth consumption measurement with different job IO sizes under various churn rates is provided in Fig. 9. The above bandwidth measurement shows the current residential broadband connections that usually come with a few hundred kbps inbound bandwidth can handle very tough computation-communication granularity applications where the per job runtime is only one minute.

Fig. 10 reports workflow runtime performance in a two stage pipeline configuration. In this experiment, we setup two groups A and B each with 9 peers, two tasks T_a and T_b each having 256 independent jobs were submitted simultaneously to these two groups. The input of T_b was set to be the output of T_a and we report the average runtime under different churn rates. The per job data IO is Class A size. The results show runtime of the second stage computation

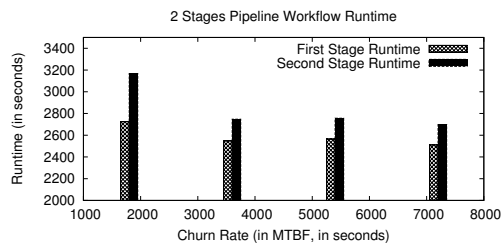


Figure 10. Two stage pipeline workflow, the runtime of the second stage is the runtime of the workflow.

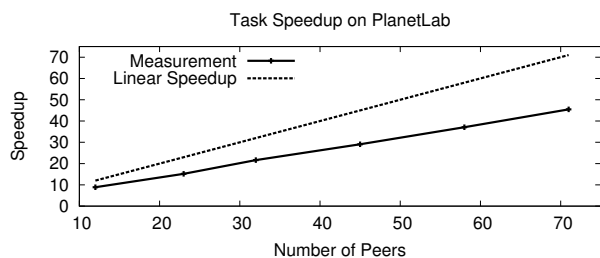


Figure 11. Speedup on PlanetLab.

is only about 10% longer than the first stage, the overall speedup for all 512 jobs is higher than 11 on 18 nodes with normal churn. Higher churn rate would impact more on the runtime but not significantly. The runtime differences between the two stages are caused by the fact that the second stage has to wait for the output of the first stage and thus causes delays.

The P2P-Tuple software has also been deployed on 72 PlanetLab [5] nodes. The scalability experiment results reported in Fig. 11 shows the speedup performance when using different number of peers compared with the optimal linear speedup. The MTBF of nodes is set to be 2 hours which is the normal MTBF observed in [4], IO size per job is Class A and the per job runtime is 2.5 minutes, other parameters are same as the ones used in previous experiments. To isolate the tail effect's influence, the number of jobs used is also set to be proportional to the number of peers in this experiment. The software exhibited a consistent efficiency of around 66% on PlanetLab, which is slightly lower than the 74% efficiency observed on cluster. This performance drop is due to the highly limited processing and bandwidth resources on some PlanetLab nodes. The results demonstrated good scalability in which the speedup increases proportional to the number of peers for up to 72 peers in such Internet environment. Our simulation results show a similar linear speedup trend on a much larger P2P network.

6 Conclusion and Future Work

In this work, the Tuple Space concept bridges the well studied P2P techniques to the emerging VC systems. The evaluation results show that with carefully designed fault tolerance mechanisms, the proposed design can handle

churn rate several times higher than the expected rate in real world deployed systems. To safeguard the reliability, the redundancy level parameter is of key importance, we plan to study the optimal redundancy level adaption to network environment like the peer churn rate.

References

- [1] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th International Workshop on Grid Computing (GRID 2004)*, pages 4–10, 2004.
- [2] D. P. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *CCGRID*, pages 73–80. IEEE Computer Society, 2006.
- [3] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), Oct. 2002.
- [4] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *J. Parallel Distrib. Comput.*, 63(5):597–610, 2003.
- [5] B. N. Chun, D. E. Culler, T. Roscoe, A. C. Bavier, L. L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: An overlay testbed for broad-coverage services. *Computer Communication Review*, 33(3):3–12, 2003.
- [6] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
- [7] D. Kondo, M. Tauber, C. L. B. III, H. Casanova, and A. A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *IPDPS*. IEEE Computer Society, 2004.
- [8] V. M. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao. Cluster computing on the fly: P2p scheduling of idle cycles in the internet. In G. M. Voelker and S. Shenker, editors, *IPTPS*, volume 3279 of *Lecture Notes in Computer Science*, pages 227–236. Springer, 2004.
- [9] M. O. Neary, B. O. Christiansen, P. Cappello, and K. E. Schauer. Javelin: Parallel computing on the internet. *Future Generation Computer Systems*, 15:659–674, 1999.
- [10] M. S. Noble and S. Zlateva. Scientific computation with javaspaces. *Lecture Notes in Computer Science*, 2110/2008:657–666, 2001.
- [11] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, 1989.
- [12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, 2001.
- [13] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX '04)*, pages 127–140, Boston, Massachusetts, June 2004.
- [14] R. Rodrigues and B. Liskov. High availability in dhds: Erasure coding vs. replication. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS05)*, Ithaca, NY, February 2005.
- [15] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.
- [16] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 188–201, Oct. 2001.
- [17] J. Tian and Y. Dai. Understanding the dynamic of peer-to-peer systems. In *Proceedings of the 6th International Workshop on Peer-to-Peer Systems (IPTPS '07)*, Feb. 2007.