# REDUNDANCY TOLERANT COMMUNICATION ON VOLATILE NODES[*]

Nagarajan Kanna, Jaspal Subhlok,Edgar Gabriel,Margaret Cheung[†], and David Anderson[‡]

Department of Computer Science
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

Technical Report Number UH-CS-08-17

December 5, 2008

**Keywords:** Dataspace, Redundancy, Volatile Systems, Linda, Desktop grids, Communication library

## Abstract

Idle PCs represent a massive platform for scientific parallel computing but they are volatile; the availability of a PC for guest computations can change suddenly and unexpectedly based on the actions of the PC owner. To execute effectively in a volatile environment, a communicating parallel program must employ checkpoint-restart and/or redundancy to make continuous forward progress in the presence of routine failures. A communication model based on one-sided Put/Get calls, pioneered by the Linda system, is a good match for such an environment as processes can execute their communication operations independently and asynchronously. However, Linda and its many variants are not designed for processes that are replicated or restarted from checkpoints. The key problem is that a single logical operation may be executed multiple times by different instantiations of the same process at different times. This paper presents the design, execution semantics, implementation, and validation of a communication layer between processes that may be replicated or re-executed. The framework is integrated with BOINC middleware for volunteer computing. Results are presented for selected benchmarks as well as Replica-Exchange Molecular Dynamics application.

[*]Partial support for this work was provided by the National Science Foundation's Computer Systems Research program under Award No. CNS-0834750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[†]Department of Physics, University of Houston

[‡]Space Science Lab,UC Berkeley

# REDUNDANCY TOLERANT COMMUNICATION ON VOLATILE NODES[*]

Nagarajan Kanna, Jaspal Subhlok,Edgar Gabriel,Margaret Cheung[†], and David Anderson[‡]

**Abstract**

Idle PCs represent a massive platform for scientific parallel computing but they are volatile; the availability of a PC for guest computations can change suddenly and unexpectedly based on the actions of the PC owner. To execute effectively in a volatile environment, a communicating parallel program must employ checkpoint-restart and/or redundancy to make continuous forward progress in the presence of routine failures. A communication model based on one-sided Put/Get calls, pioneered by the Linda system, is a good match for such an environment as processes can execute their communication operations independently and asynchronously. However, Linda and its many variants are not designed for processes that are replicated or restarted from checkpoints. The key problem is that a single logical operation may be executed multiple times by different instantiations of the same process at different times. This paper presents the design, execution semantics, implementation, and validation of a communication layer between processes that may be replicated or re-executed. The framework is integrated with BOINC middleware for volunteer computing. Results are presented for selected benchmarks as well as Replica-Exchange Molecular Dynamics application.

**Index Terms**

Dataspace, Redundancy, Volatile Systems, Linda, Desktop grids, Communication library

## I. Introduction

In recent years ordinary desktops and PCs have been employed successfully for large scale scientific computing, most commonly using Condor [16] or BOINC [3] as middleware. The Condor scheduler enables ordinary desktops to be employed for compute intensive applications. It is deployed at over 850 known sites with at least 125,000 hosts around the world. The BOINC middleware uses volunteered public PCs for scientific applications when idle. It has been remarkably successful, managing over half a million nodes and over 30 scientific research projects since its release in 2004.

Idle desktops represent a potentially immense but *volatile* resource, i.e., they are heterogeneous and their availability to guest scientific applications can change suddenly and frequently based on the desktop owner's actions. Execution of communicating parallel applications on volatile nodes is extremely challenging because failures are frequent and the failure of a single process can cause the entire application to fail. A mechanism for fault tolerance, such as checkpoint-restart or redundant replicated processes, is essential for successful execution in a volatile environment. Put/Get style asynchronous one-way communication pioneered by Linda [6] is potentially a good fit for communication on volatile nodes as it provides an abstract global shared space that processes can use for information exchange without a temporal or spatial coupling. However, redundancy implies that a variable number of instances of a single logical process may execute asynchronously, but this is not permissible in systems like Linda that provide an abstract global shared space.

This paper introduces the Volpex dataspace API for synchronous anonymous Put/Get style communication among tasks. The syntax and semantics of the API are a variant of Linda. However, the execution model and implementation ensure correct execution in the presence of multiple instances of logical processes that may execute at different times. The results with redundant processes are guaranteed to be identical to the results with normal execution with single instance for each process. Hence the framework provides robust communication between processes despite replication for fault tolerance and restart of processes from independent individual checkpoints. The management

[*]Partial support for this work was provided by the National Science Foundation's Computer Systems Research program under Award No. CNS-0834750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[†]Department of Physics, University of Houston
[‡]Space Science Lab,UC Berkeley

of redundancy is transparent: neither the framework implementation nor the process instances need to be aware of the presence or scale of redundancy and replicas can die and may be created dynamically.

This communication API is a component of the Volpex framework (Parallel Execution on Volatile nodes) that employs managed redundancy as the core mechanism to achieve seamless forward application progress in the presence of routine failures. The canonical execution model consists of two or more concurrent replicas of each process. Forward execution proceeds at the speed of the fastest replica for each process and application progress continues in the face of node failures as long as at least one replica of each process is executing. The framework plans to employ a checkpoint-restart mechanism to dynamically create replicas to replace failed ones, but this is not implemented yet. The primary goal of Volpex is to transform ordinary PCs into virtual clusters to run a variety of parallel codes.

The paper presents the design, execution model, implementation, and experimental results for the Volpex dataspace API. The framework is integrated with the BOINC software and can execute on clusters as well as idle desktops. Results presented span microbenchmarks that stress the performance of the API communication calls as well as simple benchmark codes. An implementation of *Replica Exchange Molecular Dynamics* with Volpex dataspace API was also developed and employed to demonstrate the viability of this framework for a class of applications.

## II. Volpex dataspace communication

The main goal of the Volpex project is to provide robust execution on volatile desktop nodes by employing process replication and process checkpoint-restart. The objective is that the application execution state advances with the fastest replica for each process. Execution proceeds seamlessly in case of failure of process replicas as long as at least one replica per process is executing. Processes may also save their state with checkpoints independently, and restart independently and join execution after failure. We will refer to this general scenario with multiple process replicas potentially executing at different times as *redundant execution* for brevity. The Volpex dataspace communication framework is designed to support redundant execution.

### A. Volpex dataspace API

The core API for the Volpex dataspace communication library consists of calls to add, read and remove data objects to/from an abstract global *dataspace*, with each object identified by a unique *tag* which is an index into the dataspace. The concept of a dataspace is similar to that of a tuplespace in Linda. The main calls are as follows:

`Volpex_put(tag, data)`
A *Volpex_put* call writes the data object *data* into the abstract dataspace identified with *tag*. Any existing data object with the same tag is overwritten.

`Volpex_read(tag)`
A *Volpex_read* call returns the data object that matches the *tag* in the dataspace.

`Volpex_get(tag)`
A *Volpex_get* call returns the data object that matches the *tag* in the dataspace, and then removes that data object from the dataspace.

*Volpex_read* and *Volpex_get* calls are identical except that *Volpex_get* also clears the matched data object from the dataspace. Both *Volpex_read* and *Volpex_get* are blocking calls: if there is no matching data object in the dataspace, the calls block until a matching data object is added to the dataspace. A *Volpex_put* call does not block due to the state of the dataspace. Additional calls are available in the API to retrieve the process Id and the the number of processes, and to initialize and terminate communication with the dataspace server. The full API is outlined in Table I.

### B. Execution model

The basic semantics of the communication operations are straightforward as listed in the discussion of the API above. However, managing redundant execution, as discussed in the beginning of this section, is a significant

TABLE I
VOLPEX DATASPACE COMMUNICATION API

*int volpex_put (const char* tag, int tagSize, const void* data, int dataSize)*
*int volpex_get (const char* tag, int tagSize, void* data, int dataSize)*
*int volpex_read (const char* tag, int tagSize, void* data, int dataSize)*
*int volpex_getProcId (void)*
*int volpex_getNumProc(void)*
*int volpex_init(int argc, char* argv[])*
*void volpex_finalize(void)*

| | |
|---|---|
| tag | Identifies each data object in the dataspace |
| tagSize | Number of bytes of tag |
| data | Pointer to data object being read/written |
| dataSize | Number of bytes of data |
| volpex_put() | Writes data object with the tag value |
| volpex_read() | Retrieves data object matching tag |
| volpex_get() | Retrieves & deletes data object matching tag |
| volpex_getProcId() | Returns process Id of the current process |
| volpex_getNumProc() | Returns total number of application processes |
| volpex_init() | Initialize and connect with dataspace server |
| volpex_finalize() | Releases all resources and terminates |

challenge. The key problem is that a logical call (with side effects) may be executed repeatedly or executed at a time when the state of the dataspace is not consistent with sequential execution. For example, what action should be taken if a late running process replica issues a *get* or *read* for which the logically matching data object is not available in the dataspace anymore, either because they were removed by a *get* or overwritten by another *put* ?

The guiding principle for the execution model is that the execution results with redundant execution must be consistent with normal execution. If the parallel applications is deterministic, then normal and redundant executions should give the same results. If the parallel application is non-deterministic, then redundant execution will return one possible result of a normal single process execution.

The major components of the execution model are the following:

1) *Atomicity rule:* The basic *put/read/get* operations are atomic and executed in some global serial order.
2) *Single put rule:* When multiple replicas of a process issue a *Volpex_put*, the first writer accomplishes a successful operation. Subsequent corresponding *Volpex_put* operations are ignored.
3) *Identical get rule:* The first replica issuing a *Volpex_get* or a *Volpex_read* receives the value stored at the time in the dataspace. Subsequently, all replicas issuing a corresponding *Volpex_get* or *Volpex_read* receive a value identical to the first get or the first read, independent of the contents of the dataspace at the time they are executed.

The execution model can be illustrated as follows. The fastest running replicas create a *leading front* of execution as if other replicas do not exist. The execution model ensures that *a)* the trailing replicas do not cause incorrect execution of leading replicas by corrupting the dataspace (single put rule) and *b)* trailing replicas are guaranteed to receive the same data objects for read and get calls as the corresponding leading replicas (identical get rule). All replicas execute identically as the execution of trailing replicas is identical to that of the leading replica. However, which replicas are leading or trailing can change dynamically and execution proceeds seamlessly in case of failure of any replica including the leading replica, so long as at least one replica per process exists.

## III. IMPLEMENTATION

The communication API is implemented with the client server paradigm. The processes connect to a dataspace server that services communication requests. The main challenge is in implementing the execution model with support for redundant processes. We discuss the datsapace server design followed by a discussion of some of the key implementation issues and integration with the BOINC middleware.

## *A. Dataspace server design*

In order to to maintain consistent execution in the presence of replicated processes, the implementation of the Volpex dataspace API must conform to the execution semantics discussed in Section II. The atomicity rule is satisfied by a single threaded server that processes one client request at a time. In order to satisfy the *single put* and *identical get* rules of the execution model, additional machinery is needed. Each communication call (*put, get,* or *read*) is uniquely identified by the pair: *(process_id, request_number)*, where *request_number* is the current count in the sequence of requests from a process. When a communication call is issued by a process, the *process_id* and *request_number* are appended to the message sent to the dataspace server to service the request. For replicated calls from the replicas of the same server, the *(process_id, request_number)* pairs are identical.

The server implementation maintains the current *request_number* for each process, which is the highest request number served for that process so far. The server implementation maintains two logically different pools of storage as shown in Figure 1.

- *Dataspace table:* This storage consists of the current data objects indexed with tags. The “current” is in terms of the leading front of execution (or fastest process replicas) as discussed in Section II.
- *Read log buffer:* This storage consists of data objects recently delivered from the dataspace server to processes in response to *get* and *read* calls. Each object is uniquely identified by *(process_id, request_number)*.

When a communication API call is executed in a process, a message is sent to the dataspace server consisting of the type and parameters of the call and *(process_id, request_number)* information. A request handler at the server services the call as follows:

- *Put:* If the request number of the call is greater than the current request number for the process (a new put), the data objected indexed with the tag is added to the dataspace table. If the request number of the call is less than or equal to the current request number (a replica put for which the data object must already exist on the server), no action is taken.
- *Get or Read:* If the request number of the call is greater than the current request number for the process (a new get), then i) the data object matching the tag is returned from the dataspace storage, and ii) a copy of the data object is placed in the read log buffer indexed with *(process_id, request_number)*. Additionally if the call is a *get*, the data object is deleted from the dataspace table (but retained in the read log buffer).
  If the request number of the call is less than or equal to the current request number (a replica get for which the data object must exist in the read log), the data object matching *(process_id, request_number)* is returned from the read log buffer.

The design of the dataspace server is illustrated in Figure 1. Note that the communication layer does not explicitly identify replicas but implicitly manages identical replicated requests. There is an assumption that replicas of a process behave identically. Hence non deterministic programs where execution behavior is determined by, say, time dependent external I/O operations, or externally influenced random number generation, are not supported.

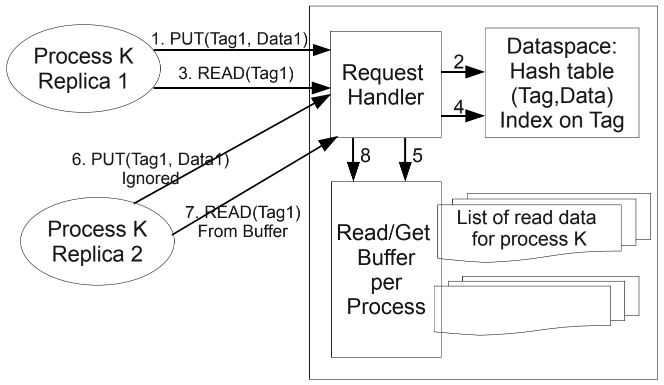

Process K Replica 1
1. PUT(Tag1, Data1)
3. READ(Tag1)
Request Handler
2
4
Dataspace: Hash table (Tag,Data) Index on Tag
6. PUT(Tag1, Data1) Ignored
7. READ(Tag1) From Buffer
8
5
Process K Replica 2
Read/Get Buffer per Process
List of read data for process K

Fig. 1. Volpex dataspace server design

### B. Implementation framework

Our communication library is built on C/C++ using TCP Sockets. The current dataspace server is a single-threaded server which multiplexes between various requests from the clients. However, the design allows a multithreaded as well as distributed implementations by partitioning the abstract global address space. The data provided by the processes is stored in-memory. The tag and data objects are stored in the form of a hash table indexed with tags. The read log buffer is implemented as a combination of hash table and lists. All data transfers are realized as one-way communication initiated by the client processes. The clients establish a connection with the dataspace server using TCP-Sockets before performing any operations. This connection is retained until all the operations on the dataspace are completed. If the connection is interrupted, processes try to reestablish the connection with the server in exponentially increasing time intervals.

The dataspace server has a circular read log buffer whose size is specified as a parameter during initialization. When the buffer is full, the oldest entry is deleted. Limited buffer size can theoretically halt the execution of a very old replica prematurely (but not of other process replicas). But with a reasonable size buffer, in practice, a very old replica is unlikely to be a factor in application progress and robustness.

### C. Integration with BOINC

The BOINC middleware developed by Anderson [3] is widely used for distributed scientific computing. BOINC offers a *bag of tasks* programming model. The clients request for work and jobs are distributed to clients for processing. The BOINC tasks have configurable expiry time after which they are considered to have failed and recreated on a different client. BOINC runs well on volatile nodes, because it offers a combination of application-level checkpointing and redundancy to handle failure and computation errors. However, the BOINC platform does not support communicating parallel programs.

This project has leveraged BOINC for management of task distribution and redundancy on volatile nodes, while applying the Volpex dataspace API for inter-task communication. When an application is compiled, it is linked with the BOINC and Volpex libraries. The BOINC redundancy mechanism is employed to create the desired degree of process replication. The user application would maintain a connection with the dataspace server that is independent of BOINC. Also, the BOINC client uses HTTP messages for communication, whereas our communication library uses TCP sockets, but this just means opening an additional outgoing port on the client.

## IV. Experiments and results

The Volpex dataspace communication library has been implemented and deployed to execute applications with replicated processes independently and within the BOINC framework. Experimentation and validation was done on compute clusters as well as ordinary desktops that constitute a "Campus BOINC" installation at University of Houston. The framework can be employed to run processes with replicas with and without the BOINC framework. However, some features, in particular, automatic restart of failed processes, is not available without BOINC. Results are presented for clients on a compute cluster for repeatability of experiments.

The framework has been tested with a variety of simple benchmark programs. This design is particularly well suited to programs with low to moderate degree and volume of communication. We selected Sieve of Eratosthenes (SoE) and Parallel Sorting by Regular Sampling (PSRS) as sample programs for performance evaluation. Results are also presented for the Replica Exchange for Molecular Dynamics (REMD), a real world application used in protein folding research that was ported to the Volpex framework. Results presented include the performance of the dataspace API calls and performance testing of the sample applications with respect to scalability, degree of redundancy and failure. The sample codes were executed on the "Atlantis" cluster which has Itanium2 1.3GHz dual core nodes with 4GB of memory running RHEL (5.1). The dataspace server was running on AMD Athlon 2.4GHz dual core with 2GB of memory running Fedora Core 5. The server and client nodes were on different subnets that are part of a 100Mbps LAN on UH campus.

### A. Benchmarking of API calls

In the first set of experiments, we recorded the time taken to execute the different API calls by the client with varying message sizes and varying degree of replication. For calls that are potentially blocking, the setup ensured that the data objects to be fetched were already available in the dataspace. For execution with replication, a worst case scenario was created where all replicas are simultaneously trying to execute the same operation.

The results for the Round Trip Time(RTT) for *put* operations measured at the client are presented in Figure 2. The effective bandwidth delivered by the server in response to *put* operations is presented in Figure 3. Note that the bandwidth presented is the aggregate bandwidth delivered by the server in response to all clients in case of replication.

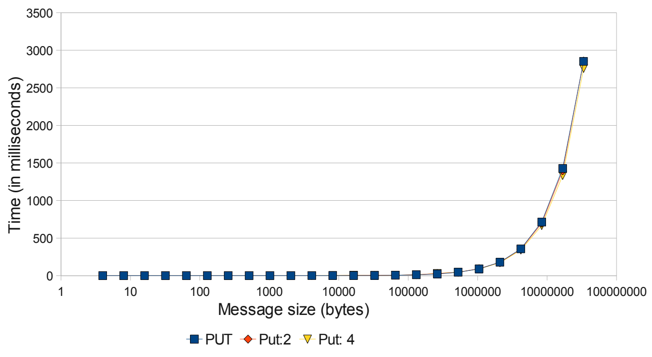


Fig. 2. Round trip time for PUT with and without replicas

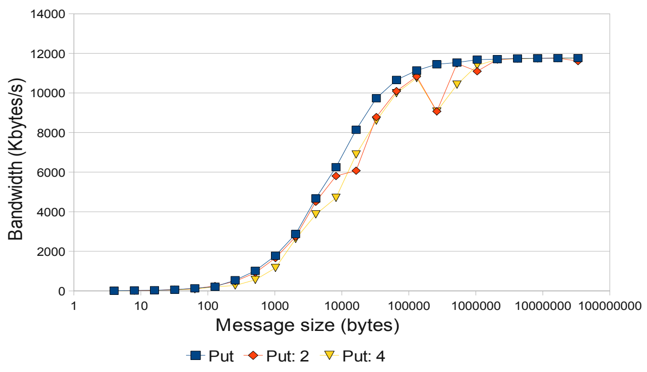


Fig. 3. Aggregate Bandwidth for PUT with and without replicas

We observe that the general trend of the RTT and bandwidth is typical of this 100Mbps LAN environment. The minimum RTT for all API operations was between .5 and .6 milliseconds which is close to the best possible on the LAN. The effective bandwidth increases with the size of the data object but flattens out around 12MBytes/sec (or 96Mbps), which is just below the network capacity of 100Mbps. Hence, the system overhead is not significant.

The figures also show the RTT and effective bandwidth with 2 and 4 replicated processes. We notice that the impact of replication is rather low: only a slight reduction in delivered bandwidth is visible for midrange of message sizes. It is instructive to recall how replicated *put* operations work. Only the first replica actually transfers the data object over the network, while calls from other replicas are returned without moving the message to the server. Thus, the network traffic does not increase significantly with replication. Hence, it is not surprising that the effective bandwidth delivered by the server is not significantly affected. The slight reduction is attributed to the overhead of processing of *put* calls from other replicas.

The results for the Round Trip Time(RTT) and delivered bandwidth for *get* operations are presented in Figure 4 and Figure 5, respectively. We omit the results for *read* operation as they are virtually identical to those for the *get* operation.

Without replication, the performance of *get* operations is very similar to the performance of *put* operations and the same discussion applies. However, the behavior with replicas is very different for *get* operations. It is instructive

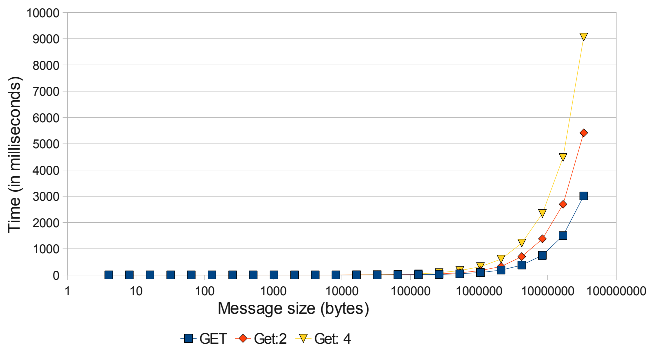


Fig. 4. Round trip time for GET with and without replicas

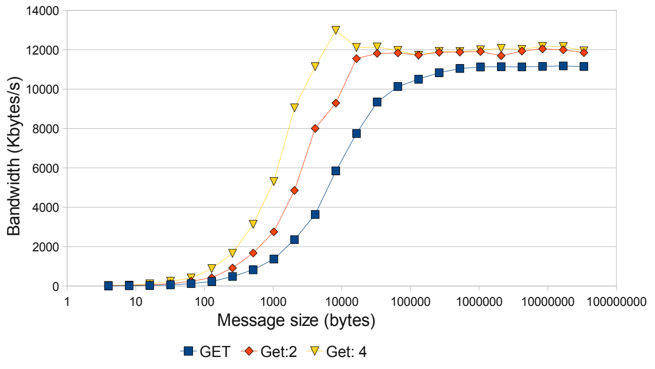


Fig. 5. Aggregate Bandwidth for GET with and without replicas

to recall that replicated *get* operations are handled very differently from *put* operations. Each replicated *get* call leads to the entire data object being transferred from the server to a client replica. Hence, for a degree of replication of $k$ the network traffic for *get* calls increases by a degree of $k$ while it remains unchanged for *put* operations.

Figure 4 shows an increase in the round trip time in the range of 25% with the addition of each replica. (This is also true for cases where the difference is not visible in the graph) The increase in RTT with replicas is explained by the increased network congestion and increased load on the dataspace server. At the same time, Figure 5 shows that the aggregate bandwidth delivered by the server *increases* significantly with replication except for very high message sizes where the bandwidth is (probably) limited by the network capacity. The server is able to register a higher bandwidth as 2 and 4 replicas imply that the aggregate rate at which the data is being demanded by the clients increases by a factor of 2 and 4, respectively.

## *B. Example Programs*

We used two example programs for testing: PSRS and SoE. The Parallel Sorting by Regular Sampling (PSRS) sorts distributed data sets by regular sampling of the data for good pivot selection. The Sieve of Eratosthenes (SoE) is a well known algorithm for finding all prime numbers up to a specified integer. PSRS was used to sort 64 million elements and SoE to identify primes up to 8 billion.

Figures 6 and 7 present the execution time for PSRS and SoE. Both programs scale well up to 32 processes, after which the execution time stays the same or increases slightly. For basic programs with modest data sizes that are not optimized for parallelism, we consider the performance and scalability acceptable for the execution environment. Basic communication optimizations are likely to improve scalability as well as performance with replication (discussed next) significantly. For example, the SoE program sends one number in each communication operation and blocking can reduce the communication cost significantly.

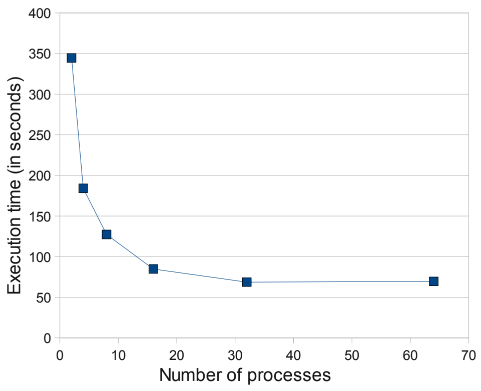


Fig. 6. Performance of Parallel Sorting by Regular Sampling (PSRS)

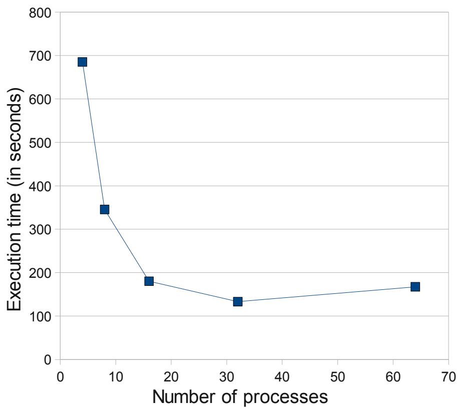


Fig. 7. Performance of Sieve of Eratosthenes (SoE)

Process replication is our approach to overcoming the volatile nature of nodes. For investigating the impact of process replication on performance, we measured the execution time with number of replicas varied from 1 to 5. PSRS sorts 64 million elements with 16 processes and SoE counts up to 2 billion with 16 processors for this set of experiments. Figure 8 presents the results for PSRS and SoE.

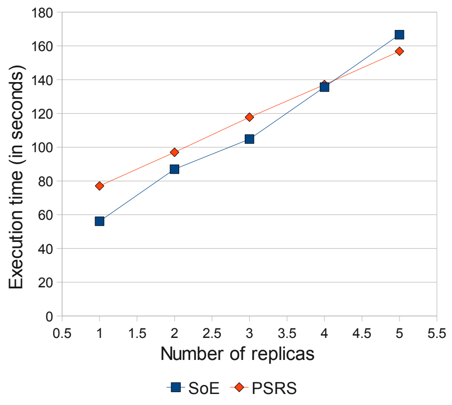


Fig. 8. Redundancy results for PSRS and SoE

We see a modest but steady linear increase in execution time with increasing degree of redundancy for both programs. The reasons can be understood in the context of benchmarking results discussed earlier that show that the execution time for *get* and *read* operations can increase significantly, primarily because the network traffic and the workload on the server is increased. The results highlight the fact that the level of redundancy should be selected carefully as it does have some negative impact on application performance beside increasing resource usage. Typically a redundancy level of two is sufficient for many scenarios. The results also highlight the potential increase in performance with a distributed implementations of the dataspace server.

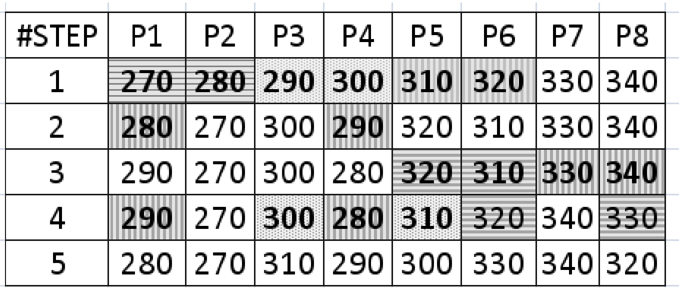
We also evaluated the impact of node failures on application performance. I all cases, the application execution time was not negatively impacted with failure of any number of replicas. This is the expected behavior for homogeneous nodes. Of course, if all replicas of a process fail, then the application will fail.

## *C. Application: Replica Exchange for Molecular Dynamics*

The Replica Exchange Molecular Dynamics (REMD) formulation [15] tries to overcome the multiple-minima problem by exchanging the temperature of non-interacting replicas of the system running at several temperatures. The context in which one of the authors of this paper (Cheung) is applying REMD is “crowding” in cell like environments [9], [14]. In REMD, each node runs a piece of molecular simulation at a different temperature using the AMBER program [7]. The number of nodes to simulate the system depends on the system’s size and types of interactions. At certain time steps, communication occurs between neighboring nodes. An exchange is initiated based on the Metropolis criterion, in case a given parameter is less than or equal to zero. In our case, the data exchanged is the temperature.

In our implementation of this code, the dataspace API is used to i) store process-temperature mapping, ii) synchronization of the processes at the end of each step, iii) identification and retrieval of energy values from neighboring processes, and iv) swapping of temperatures between processes when needed. Currently REMD runs on clusters and data exchange is performed using a shared file system and file locking.

In the REMD experiments we conducted, each replica represents a process which starts running simulations for a certain temperature. At the end of each step, neighboring temperatures may be exchanged between processes based on the Metropolis criterion. A snapshot of such exchange of temperatures is presented in figure 9, from one of the experiments. It shows how temperatures are exchanged for an execution with 8 replicas and 5 steps. In each step, temperatures that are swapped are highlighted in bold with the same background pattern.



| #STEP | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 |
|---|---|---|---|---|---|---|---|---|
| 1 | **270** | **280** | **290** | **300** | **310** | **320** | 330 | 340 |
| 2 | **280** | 270 | 300 | **290** | 320 | 310 | 330 | 340 |
| 3 | 290 | 270 | 300 | 280 | **320** | **310** | **330** | **340** |
| 4 | **290** | 270 | **300** | **280** | **310** | 320 | 340 | 330 |
| 5 | 280 | 270 | 310 | 290 | 300 | 330 | 340 | 320 |

Fig. 9. REMD - Temperature(K) swaps for 8 replicas with 5 steps

Figure 10 presents comparison of application execution time between the dataspace approach versus the current approach (PERL & File locking). We observe that the dataspace API does reduce the execution time of the application to some extent. Since the core process execution is identical in both cases and accounts for most of the execution time, the improvement in the communication component is significant although it is difficult to isolate. The most significant result is that the new approach allows this application to run reliably on volatile ordinary desktops because of fault tolerance.

## V. Related work

Idle desktops are widely used for parallel and distributed computing. The Berkeley Open Infrastructure for Network Computing (BOINC) [3] is a middleware system widely used for volunteer computing where people donate the use of their computers to help scientific research. Condor [16] is a specialized workload management system for

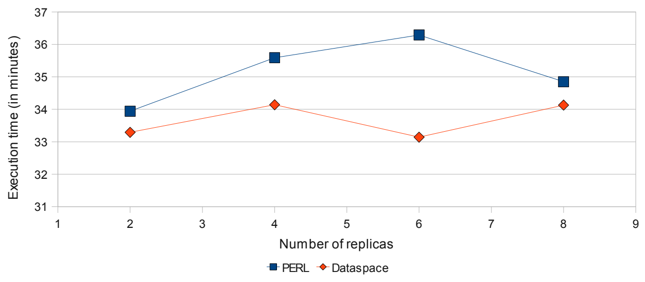


Fig. 10. REMD - Execution time using PERL/files versus Dataspace API

compute-intensive jobs. Condor can effectively harness wasted CPU power from otherwise idle desktop workstations. Other systems that build desktop computing grids include Entropia [10] and iShare [13]. A critical challenges in running parallel programs on ordinary desktops is volatility. The BOINC system uses a combination of redundant computation of tasks and application level check pointing to overcome volatility. Condor uses checkpointing [11] to handle volatility. In many circumstances, when Condor detects that a machine is no longer available (such as a key press detected), it is able to transparently produce a checkpoint and migrate the job to a different machine which would otherwise be idle. While these mechanisms are valuable for long running sequential and bag-of-task codes, they are generally not sufficient for communicating parallel programs.

Several implementations of the MPI library have developed support for fault tolerance with checkpoint-restart. Perhaps closest to our work is MPICH-V [5] a system that offers scalable fault tolerant MPI for volatile nodes based on uncoordinated checkpoint/roll-back and distributed message logging. The library, however, is not designed for redundant execution.

Linda [6] is a model of coordination and communication among several parallel processes based on a logically global associative memory, called a tuplespace, in which processes store and retrieve tuples. The work on fault tolerance in Linda has mostly focused on making operations atomic and providing a tuplespace that survives failure [4]. IBM also has a tuplespace platform named TSpaces[2]. JavaSpaces [12] is an implementation of Linda in Java by Sun Microsystems, incorporated into the Jini project. Scalable Asynchronous Replica Exchange for Parallel Molecular Dynamics Applications (Salsa) [17] is Linda adaptation for Molecular Dynamics. It is a decentralized and asynchronous realization of the Replica Exchange algorithm for simulating the structure, function, folding, and dynamics of proteins. Publish/subscribe [8] (or pub/sub) is an asynchronous messaging paradigm where senders (publishers) of messages are programmed to send their messages to specific receivers (subscribers). One well known implementation of pub/sub is Java Messaging Service [1] developed by Sun Microsystems.

The design of Volpex dataspace API has borrowed from these projects. However, the support for routine replicated operations is unique. In a volatile environment, system mean time between failure (MTBF) can drop to minutes or seconds for even 10s to 100s of nodes. In such scenarios, the checkpointing approach becomes impractical and replication approach more appealing [18].

## VI. Conclusions and future work

This paper introduces the Volpex dataspace API that allows efficient Put/Get operations on an abstract shared global shared memory in the presence of redundant execution. An innovative communication model and implementation ensure consistent execution results despite multiple invocations of communication operations at different times by independently executing process replicas.

The target of this research is the Volpex execution environment that aims to support efficient execution of communicating parallel programs on volatile idle desktops. It employs process replication as the core mechanism for fault tolerance. The experiments and results demonstrate that the framework has low overhead and delivers the performance and scalability expected on LAN connected nodes, while also providing significant robustness.

While dedicated compute clusters will always be preferable for many latency sensitive applications, other loosely coupled parallel applications can gain reasonable performance on ordinary desktops. However, the volatility of desktops is a central problem and the current usage of volatile desktops is limited to embarrassingly parallel (or bag of tasks) or master-slave applications, typically using BOINC or Condor middleware. We believe this work expands the realm of computing on idle desktops to a much larger class of parallel applications. If a substantial fraction of HPC applications could be executed on shared desktops, the impact will be significant as the demand for dedicated clusters will decrease and the clusters can be dedicated to latency sensitive applications that they are designed for.

The implementation used for this work is a prototype that is being improved to reach the point of public release. The system is designed to use the main memory of the dataspace server for storage but data centric applications will require seamless use of the filesystem efficiently. Distributed implementations of the dataspace server are important for server side fault tolerance as well as scalability. More applications and more experimentation on idle desktops is needed to make firm conlusions about the value of this work. Finally, additional API operations, in particular, non blocking reads that return immediately when matching data objects are not available, and single assignment puts that cannot be overwritten, are being studied for their value in application development and ease of implementation.

## REFERENCES

[1] http://java.sun.com/products/jms/. V

[2] http://www.almaden.ibm.com/cs/tspaces/. V

[3] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society. I, III-C, V

[4] D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, 1995. V

[5] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: toward a scalable fault tolerant mpi for volatile nodes. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. V

[6] N. Carriero and D. Gelernter. The S/Net's Linda kernel. *ACM Trans. Comput. Syst.*, 4(2):110–129, 1986. I, V

[7] D. Case, D. Pearlman, J. W. Caldwell, T. Cheatham, W. Ross, C. Simmerling, T. Darden, K. Merz, R. Stanton, and A. Cheng. *Amber 6 Manual*. 1999. IV-C

[8] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003. V

[9] D. Homouz, M. Perham, A. Samiotakis, M. S. Cheung, and P. Wittung-Stafshede. Crowded, cell-like environment induces shape changes in aspherical protein. *Proceedings of the National Academy of Sciences*, 105(33):11754–11759, 2008. IV-C

[10] D. Kondo, M. Taufer, C. Brooks, H. Casanova, and A. Chien. Characterizing and evaluating desktop grids: an empirical study. *Proceedings. 18th International Parallel and Distributed Processing Symposium*, pages 26–, April 2004. V

[11] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997. V

[12] M. S. Noble and S. Zlateva. Scientific computation with javaspaces. In *HPCN Europe 2001: Proceedings of the 9th International Conference on High-Performance Computing and Networking*, pages 657–666, London, UK, 2001. Springer-Verlag. V

[13] X. Ren and R. Eigenmann. iShare - Open internet sharing built on peer-to-peer and web. In *European Grid Conference*, Amsterdam, Netherlands, Feb 2005. V

[14] L. Stagg, S.-Q. Zhang, M. S. Cheung, and P. Wittung-Stafshede. Molecular crowding enhances native structure and stability of / protein flavodoxin. *Proceedings of the National Academy of Sciences*, 104(48):18976–18981, 2007. IV-C

[15] Y. Sugita and Y. Okamoto. Replica-exchange molecular dynamics method for protein folding. *Chemical Physics Letters*, 314:141–151, 1999. IV-C

[16] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005. I, V

[17] L. Zhang, M. Parashar, E. Gallicchio, and R. M. Levy. Salsa: Scalable asynchronous replica exchange for parallel molecular dynamics applications. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 127–134, Washington, DC, USA, 2006. IEEE Computer Society. V

[18] R. Zheng and J. Subhlok. A quantitative comparison of checkpoint with restart and replication in volatile environments. Technical report, University of Houston, 2008. V