

π -Spaces: Support for Decoupled Communication in Wide-Area Parallel Applications

Philip Chan

David Abramson

MESSAGE Laboratory

Faculty of Information Technology, Monash University

900 Dandenong Road, Caulfield East, VIC 3145

Australia

{pchan,davida}@csse.monash.edu.au

Abstract

Wide-area distributed systems like computational grids are emergent infrastructures for high-performance parallel applications. On these systems, communication mechanisms have to deal with many issues, including: private networks, heterogeneity, dynamic resource availability, and transient link failures. To address this, we present π -Spaces, a shared space abstraction of typed pipe objects. These objects, called π -channels, are asynchronous pipes that combine streaming and persistence for efficient communication while supporting spatial and temporal decoupling. This feature allows π -channels to be written even in the absence (or failure) of the reader. In this paper, we present the design of π -Space runtime system and provide some throughput evaluation results with our experimental prototype.

1. Introduction

In recent years, wide-area distributed systems like computational grids have become the emergent infrastructures for high-performance parallel computing. The cost-effectiveness of commodity clusters and NOWs make them well-suited as building blocks of larger wide-area distributed systems. On these multi-cluster systems/grids, the usage is typically single or parallel job submissions to individual resources. For example, a parametric sweep application may generate hundreds of jobs which are independently submitted to resources. With Grid workflows, each stage is submitted as a separate job by a workflow scheduler. In some workflows, co-allocation [4] to accommodate some stages may be necessary. And most parallel (MPI) applications are submitted as parallel jobs to individual

clusters/supercomputers.

Scientific applications that currently run on these systems fall under one of these classes: embarrassing parallel programs (including parametric sweeps and high-throughput applications), bag-of-tasks applications, and master-worker applications. However, we envision opportunities for more complex parallel applications (i.e., those that involve inter-task communications) to be executed on these wide-area systems, utilising nodes across multiple clusters.

The goal of our ongoing research is to develop a programming model to realise this vision. A key aspect of this programming model is the mechanism for communication between components of these applications. By examining these systems, we found that current wide-area multi-cluster grid systems are characterised by the following:

1. *Non-Uniform Network Quality-of-Service.* The network QoS here refers specifically to three aspects: (a) link reliability; (b) latency; and (c) bandwidth. Communication between nodes of a single cluster is typically provided by reliable, low-latency, high-bandwidth dedicated interconnects (e.g., Myrinet, Infiniband). On the other hand, the inter-cluster communication links are more likely to suffer from failures, high latencies, and fluctuating bandwidths.
2. *Support for Dynamic Joining/Leaving of Processes.* The large-scale and decentralized nature of computational grids make them highly dynamic. Nodes that participate on desktop grids [13], for example, may join and leave at will. Clusters typically use a batch queueing system, so a job submission may not lead to an immediate allocation of requested compute nodes. On heavily-utilized clusters, parallel jobs that request many CPUs may wait in

the queue for hours before getting executed. Furthermore, co-allocation [4] under these conditions of heavy load, may be difficult to achieve. Hence, it would be useful if the programming model allows an application to begin execution when some nodes are allocated and accommodate new nodes as they become available.

3. *Multiple Administrative Domains.* Computational grids are usually constructed as part of a multi-organizational collaborative effort. Often, each organization retains full administrative control over the resources that it contributes to the grid. Private networks and security policies implemented in firewalls can complicate direct communication.

In this paper, we present a programming model and runtime system that are specifically adapted to the unique characteristics of these systems. Its key feature is decoupled communication through persistence. Decoupled communication is best exemplified in the Linda tuplespace model [7], where communication between processes takes place via a logically shared space of tuples. This space serves as an insulating layer to decouple processes in time and space. Time decoupling means that communication can take place even when processes are not concurrent. Space decoupling, on the other hand, enables communication without direct identification of the target receiver.

However, efficient communication is difficult to achieve when all messages passed between processes has to go through a shared tuplespace. We propose combining the benefits of efficient pipe-based communication with the abstraction of distributed shared spaces. The abstraction of the stream or pipe is considered useful for communication in grid applications [14]. And recently, pipe-based mechanisms have been introduced for multi-cluster and grid communication. For example, DP [11, 18] provides a similar pipe-based communication interface. Taskspaces [20, 19] combine direct communication mechanisms and Linda-like tuple spaces. The main difference of our approach from these is the persistence property which allows further decoupling of communication.

2. π -Spaces: Basic Programming Model and Semantics

Communication in the π -Spaces programming model is through persistent named pipe objects called π -channels. We built on the basic pipe model was because of its conceptual simplicity and intuitiveness. At its simplest form, a pipe represents a uni-directional communication channel between a process pair. With

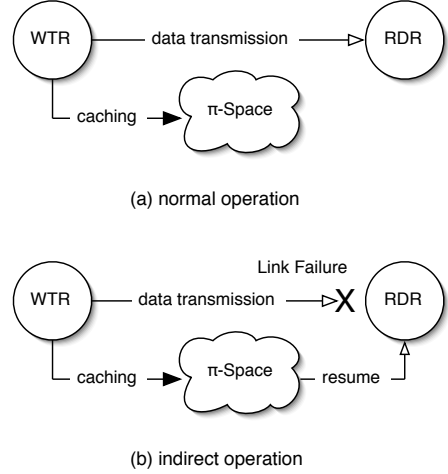


Figure 1. π -channel behavior under two conditions: (a) when link is available and reader process (RDR) is able to receive; and (b) during a link failure, writer (WTR) continues to transmit data to the space.

its underlying theoretical underpinnings in CSP [8] and other process calculi, channel-based communication has been successfully used in many systems like Occam [12] and Fortran-M [5].

2.1. π -channels: Persistent Named Pipes

In the π -Spaces model, the basic unit of communication is the π -channel. Each π -channel is an enhanced uni-directional pipe that has a string name (process and location-independent), a set of attributes and a data-stream payload for efficient bulk data transport. This payload is an FIFO-ordered sequence of data elements which may be typed or untyped. An untyped π -channel resembles a Unix file with a simple byte-stream structure. Typed data elements are useful for communication between heterogeneous systems, e.g., across multiple clusters.

Unlike conventional pipes, π -channels may be created and written to at any time, independently of the presence or status of the reader. This asynchronous operation encourages temporal decoupling between readers and writers. Persistence is achieved by transparently storing the π -channels into π -Space. When the reader arrives, it can lookup the π -channel from the space and retrieve the associated data stream.

During the creation of a π -channel, when the matching reader is known and available, a connection (direct, if possible) is employed to efficiently transfer the payload data from the writer. Effectively, the π -channel

payload data is transmitted like a TCP stream to the receiver. At the same time, the delivery of the π -channel to the π -Space continues (Figure 1a). This “caching” mechanism makes it possible for writing to proceed even if the link and/or reader has failed in the middle of communication. At this point, the π -channel data will be directed towards the associated π -Space (Figure 1b). When the link is eventually restored, the reader may resume from the last point by retrieving the cached copy.

For simplicity, we opted for uni-directional mode of communication. Bi-directional behavior can be easily implemented using a pair of π -channels. The basic π -channels programming interface consists of the following primitives:

1. `pi_attach(str n, int sp, int blk)`
attaches to a π -channel with the given name `n` on space identified by `sp`, if it does not exist, the caller may be blocked if `blk` is set. On success, it returns a descriptor representing the “read” end of the π -channel.
2. `pi_create(str n, int sp)`
creates a π -channel with the given name `n` on space `sp`. On success, it returns a descriptor representing the “write” end of the created π -channel.
3. `pi_read(int d, ptr b, size_t c, dtype_t1 dt)`
reads `c` elements of type `dt` into the memory block with starting address specified in `b` from the descriptor `d` that represents the “read” end of the attached π -channel.
4. `pi_write(int d, ptr b, size_t c, dtype_t dt)`
writes `c` elements of type `dt` from the memory block with starting address specified in `b` into the “write” end of the π -channel, represented by the descriptor `d`.
5. `pi_close(int d)`
closes the “write” end of the π -channel represented by the descriptor `d`.
6. `pi_detach(int d)`
closes the “read” end of the π -channel represented by the descriptor `d`.

The `pi_attach()` and `pi_create()` are complementary operations, with matching performed according to the π -channel name. This name is an alphanumeric string that is provided by the programmer and is used for exact matching during attach and create operations. If two or more channels with the same name are created, they will be considered as distinct

¹The `dtype_t` declaration supports various data types that are commonly used, e.g., `PI_INT`, `PI_FLOAT`, etc. The details of this is omitted for brevity.

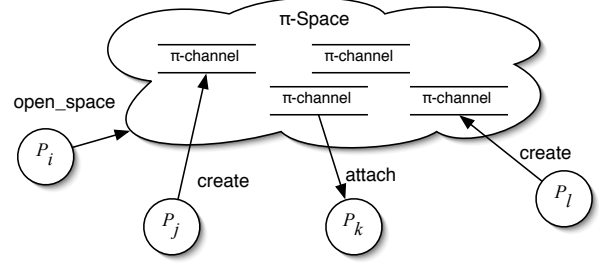


Figure 2. Decoupled communication using π -channels between processes through a single π -Space.

channels and will be attached in FIFO order. The `pi_attach()` will always attempt to retrieve an unread π -channel. If none exist, a previously read π -channel will be retrieved. If a previously read π -channel is to be retrieved, a `pi_attach.any()` primitive has to be invoked.

Implicit Multicast Semantics. The behavior of the `pi_read()` ensures that π -channel data is marked as read but not deleted from the pipe. This allows fault-tolerant communication, for example, when an application that was reading an open π -channel crashes, it can resume from the last point. This also means that all π -channels can be read by multiple sinks, provided that they use the same name during the `pi_attach()` and that only one π -channel has that name.

2.2. π -Spaces: Spaces Abstraction for Shared Pipes

Distributed processes communicate and coordinate with each other through one or more π -Spaces. Like tuplespace, a π -Space is an abstraction of shared memory for structured objects. These objects are π -channels, which are explicitly posted and retrieved by processes (Figure 2).

The model of persistent π -channels fits elegantly with shared spaces. When a π -channel is created, a copy of it is automatically posted on the specified π -Space. This means that creation of π -channel is always a non-blocking operation. During a `pi_attach()`, the π -Space is accessed to retrieve the named π -channel. If there are several π -channels with the same name on the space, the `pi_attach()` will retrieve the oldest one.

Designing this primitive to use wildcards to attach to multiple π -channels was considered. But for simplicity, we decided to restrict this primitive for accessing

only one π -channel object. Limiting to exact string matches in the names also leads to an efficient implementation of the lookup mechanism.

π -Spaces are logical name spaces that can be created and destroyed at runtime. Two simple API functions are provided for managing π -Spaces.

1. `int pi_open_space(str n, str lookup_ip_port, str cache_ip_port)`

This primitive performs π -Space creation and initialisation, where the `lookup_ip_port` and `cache_ip_port` arguments specify the machine address (IP) and port number of the lookup and the cache server (respectively). If the π -Space with the given `n` does not exist yet, it is created and made active (usable) for the calling process. If it already exists, it is simply made active. The atomicity property of this primitive ensures that whenever two or more processes attempt to create and activate a space, only one π -Space will be created. On success, this primitive returns a unique π -Space identifier.

2. `int pi_close_space(int s)`

This primitive is called by processes to indicate shutdown of a given space specified by its unique identifier. Once the last process has performed this primitive, the corresponding π -Space will be deleted, unless it is not empty.

2.3. Usefulness of π -Spaces/ π -channels

The combination of π -channels and π -Spaces presents a decoupled communication mechanism for writing applications. By employing user-specified string names to identify π -channels, anonymous communication between processes is made possible. We believe this ability to name objects in a location transparent way is useful for dynamically coupling processes on a wide-area distributed environment.

To illustrate this, we present two example applications. In a multi-body (or particle) simulation, the physical space containing the particles may be logically partitioned among a set of processes. Each process computes the forces and velocity of the particles. At the end of each iteration, particles that are no longer inside the assigned space of the process are sent to other processes. π -channels may be used for this communication. A convention may be used to name π -channels so that processes need not explicitly identify each other for communication. For example, the process in charge of region (1, 1, 1) may send particles to a neighboring region (1, 1, 2) at the end of iteration 10 using a π -channel with a name of "particles_10-1-1-1_to_1-1-2".

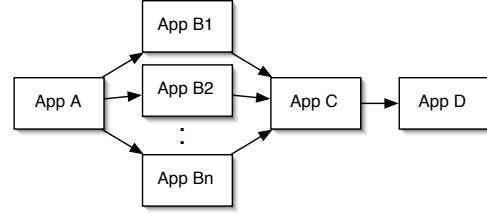


Figure 3. A hypothetical workflow application.

π -channels may also be used for coupling scientific workflows. A workflow (e.g., Figure 3) consists of multiple application components that communicate with each other. Each component reads some input (e.g., files, data streams, etc.) and generates one or more outputs. Coupling these components via π -channels will require agreement on a naming convention for the pipe objects. Since π -channel operations are asynchronous (i.e., write operations may proceed independently of the reader/s), flexible scheduling these components is made possible. For example, if resource availabilities do not permit App D to be scheduled concurrently with App C, the output of the latter may be stored using a π -channel. When App D is started, it can proceed from reading the output from the corresponding π -channel.

3. Design and Implementation

The π -Spaces runtime system is designed around the following goals:

1. *Support for Multi-Cluster Configurations.* π -channels are intended to provide as a communication mechanism for processes running on nodes of different clusters. These nodes are usually configured to use private IP addresses, making them inaccessible from outside the cluster. In our π -channels implementation, cache servers serve as proxies to enable these nodes to communicate with each other.
2. *Scalability.* Implementing the π -Spaces abstraction requires some form of lookup service. This service is realized using a collection of daemon processes that are deployed on the multi-cluster environment. For scalability, a large application may use multiple π -Spaces, each π -Space is managed by a single lookup service. This concept is similar to localities in the X-KLAIM [2].
3. *Decentralized Control.* The runtime system is implemented as a collection of daemon processes. However, the design employs a decentralized structure which ensures no single point of failure. This

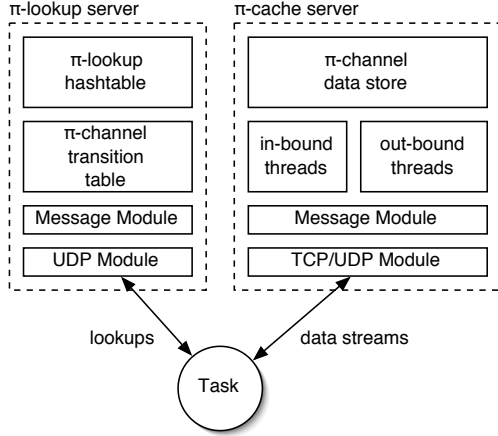


Figure 4. The internal components of π -lookup and π -cache servers.

means that the lookup service and the cache service are independent and may be setup on different machines for flexibility.

3.1. Overall Architecture

In order to realise those design goals, we designed and implemented the runtime system as consisting of multiple servers. We split the functions needed to support the shared space into two key daemon processes: the π -channel-lookup service; and the π -channel-cache service. Figure 4 shows the internal components of these two daemon processes. Each π -Space is implemented using a pair of these daemon processes, hence it is possible to employ multiple π -Space to match the structure of the underlying resources.

3.2. π -channel Lookup Service

The lookup service implements a hash table for matching the names of π -channels as they are created and attached to. Since the objects matched are actually streams, where reading and writing has to be synchronised, the lookup service is implemented using a state transition mechanism (see Figure 5). We employed an event-state table to determine the appropriate action to perform in response to a given message (e.g., ATTACH, CREATE, READ, etc.) given the current state of the π -channel. For example, this mechanism prevents reading beyond the (current) end of the stream, providing some simple flow control.

When a process issues a `pi_attach()` or `pi_create()`, a UDP message is sent to the applicable lookup server. This lookup server unpacks

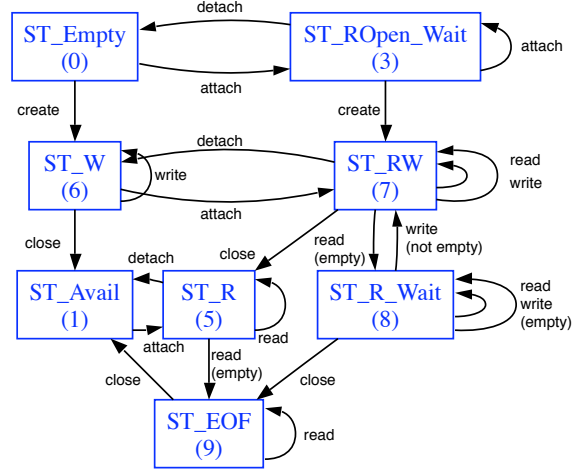


Figure 5. π -channel state transition diagram.

this message and determines the appropriate response to the process. For example, a `pi_create()` will receive a reply to allow the process to subsequently send (a) the stream data to the intended reader; and (b) send this data to the appropriate cache server.

We present a short description of the state transition diagram. `ST_Empty` represents the initial state of a π -channel, i.e., it does not exist yet. If a writer has performed a `pi_create()`, this π -channel will be created, and subsequent `pi_write()` will be allowed (see `ST_W`). If no reader arrives to attach to this stream, the writer will eventually complete and close the stream, leading to `ST_Avail`. At this state, any `pi_attach()` will be directed to receive the stream data from the cache server, since the writer has completed the π -channel. State `ST_ROpen_Wait` represents a π -channel that is waiting for its creation, blocking the `pi_attach()` operation. Eventually, `ST_RW` is reached with `pi_create()`, allowing both reader and writer to proceed. If reading is faster than the writing, the `ST_R_Wait` state may be reached.

3.3. π -Space Cache Service

In Figure 4, we also present the internal design of the cache server. Each cache server maintains a thread pool for managing incoming and outgoing transmissions. Essentially, this server behaves like a non-archival file storage server, where π -channel data are buffered temporarily. Multiple threads are employed (the number of which is configurable), to provide the necessary concurrency to support multiple readers and writers. The `inbound threads` handle incoming data streams from writers. The `outbound threads` perform the forwarding of streams to intended readers.

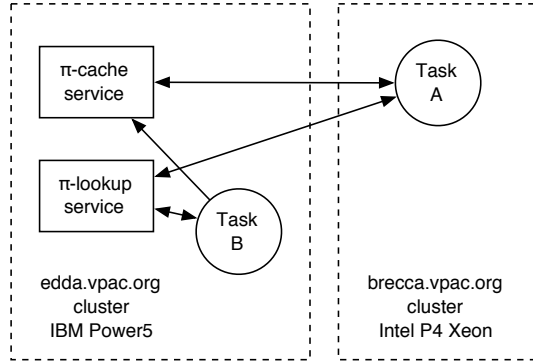


Figure 6. Experimental testbed.

The cache server also doubles as a proxy service. To enable this functionality, the cache server may be setup to run on a head-node of a cluster. This allows other nodes within the cluster to directly communicate with it. Since head-nodes often have public IP-addresses, they can also be accessed by other machines.

4. Experimental Results

We have implemented a prototype of this runtime system in C and using TCP/UDP for efficient communication. The API is also implemented providing the basic functionality presented in the previous section.

To evaluate the performance of our prototype, we conducted tests to measure the data transfer throughput when the cache server is used. Our testbed consists of two clusters: `edda.vpac.org` – a 144-cpu IBM Power5 cluster; and `brecca.vpac.org` – a cluster of Intel Pentium 4 Xeon processors. Figure 6 presents the configuration of our testbed, where the cache server and lookup server are run on `edda.vpac.org` and one process on each cluster.

We built a ping-pong benchmark program involving reading and writing multiple pairs of π -channels. At each iteration, the application performs two phases: (a) one process creates a π -channel and the other attaches to it; and (b) vice-versa. Figure 7 presents the results of our tests. The x-axis presents the 6 different sizes used in performing `pi.write()` operations. Overall, each run involved 10 iterations creating a total of 20 π -channels. For each π -channel created, the processes sent each message 100 times. For example, when the message size we used was 5,000 bytes, the total data transferred along one direction was 5,000,000 bytes.

In Mode A, we have enabled all API functions to involve synchronisation messages with the lookup server. For example, the `pi.read()` and `pi.write()` operations involved communication with the lookup server.

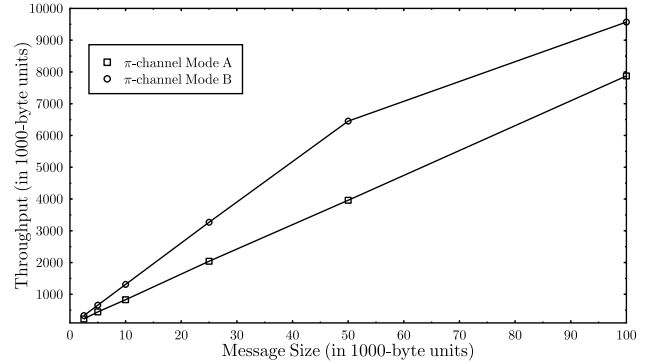


Figure 7. Measured throughput from our synthetic benchmark. 10 rounds, each round using a π -channel-pair for bi-directional transfers.

In Mode B, we optimised our implementation so that synchronisation is controlled by the corresponding processes that are directly involved in the communication. These reduced the need to send superfluous messages to the lookup server and improve the effective throughput.

The results show that when message sizes are small, bandwidth is affected by the overheads due to the synchronisation messages between the lookup server, the cache server, and the processes. This means that large reads and writes perform better as these synchronisation costs are covered by the data transfer costs. It is to be noted that frequent `pi.attach()` and `pi.create()` operations also incur overheads on the lookup server.

We also conducted a test to determine how our implementation compares with a socket-based benchmark using TCP streams for data transfers. Unlike our previous test, this π -channel version used only a single pair of π -channels for coupling the two processes. The testbed is also configured as in Figure 6. We temporarily disabled the cache server, so that it will not be involved to store the stream payload. The TCP version uses a pair of processes, one setup to listen to a port for incoming connections, and the other configured to connect to the port. The π -Space code is simpler as there is no need to configure IP addresses and port numbers. The results are encouraging (Figure 8), showing comparable performance between our implementation (using Mode B – with no extra synchronisation messages with the lookup server) and that of directly using TCP sockets.

On the TCP results, we observed a drop in throughput when messages reached 50,000 bytes long. We suspect that this is due to TCP’s mechanism to re-

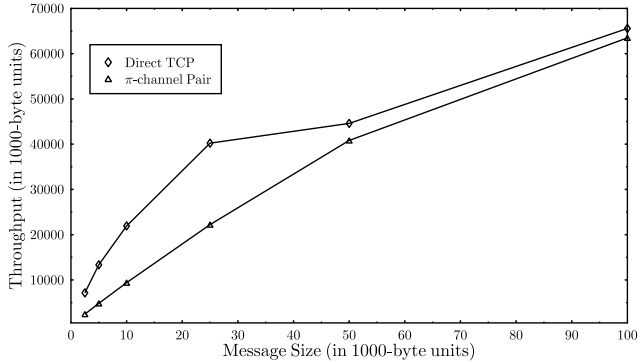


Figure 8. Measured throughput compared to that of TCP. On the π -Space version, we used a pair of π -channels. The TCP version used a single TCP connection for bi-directional transfers.

duce congestion by throttling the data flow rate once a certain threshold is reached. We were unable to confirm this as packet traces can not (and should not) be done on a production facility. Although the π -channels version is built on TCP, it employs additional headers and synchronisation messages, translating to overheads that yields a slower data transfer rate.

5. Related Work

Linda’s tuplespace model [7] is perhaps one of the novel ideas of its time. The model is simple and elegant, allowing for decoupling of communication between processes. Current variants of Linda include: IBM TSpaces [21] and Sun Microsystems JavaSpaces [6]. Both are based on the tuple and tuplespace concept and are implemented for Java.

Communication using the channel abstraction has its early beginnings in Hoare’s Communicating Sequential Processes (CSP) [8]. Occam [12] is perhaps one of the first programming languages based on CSP. Channels in CSP provide two basic operations: input (?) and output (!). Because of the simplicity, this channel concept is widely adopted in current process calculi. There are also many coordination languages that use channel or similar abstractions including DP [11], Darwin [15], POLYLITH [17], and Conic [16].

The idea of using persistence in communications has been proposed before. These include: persistent pipes that ensures transactions commit [9]; persistent connections (implemented over TCP) that allows endpoints to relocate without losing data [22]; and more

recently, the persistent streaming protocol that guarantees eventual delivery of data to receivers [10]. Unlike π -Spaces, these systems do not provide a logical addressing space to simplify coupling of communication endpoints. For example, IP addresses and ports are necessary for configuring endpoints in both [9, 22].

Taskspaces [20, 19] is a system that bears the closest resemblance to π -Space and π -channels. It uses Linda tuplespace model combined with direct communication for performance but does not provide the persistence during communication.

π -Spaces/ π -channels is motivated from our previous work Netfiles and Griddles. In [3], we have proposed the use of overloading conventional file I/O primitives with IPC functionality for parallel programming on clusters and NOWs. π -channels extend this abstraction to provide persistence and incorporate a logical naming space called the π -Space. Griddles [1] provide mechanisms to transparently multiplex I/O operations, enabling legacy applications that reads and writes conventional files to be coupled and executed over a Grid infrastructure.

6. Conclusions and Future Work

To the best of our knowledge, π -Spaces/ π -channels is the only programming model at this time that combines channels/pipes with shared space concept (for logical addressing and persistence). Our model enables decoupled communication between processes, allowing a writer to proceed even in the absence of the reader. This relaxes the need for both reader and writer to be executing concurrently, as data can be directed and stored temporarily on a shared space. The shared space presents a logical addressing space, enabling easy coupling between processes. At the same time, the persistent channel property is a natural extension of the idea of storing tuples in tuplespaces.

These properties address communication issues in grids and wide-area distributed systems: (a) communication costs can vary greatly between resources and can suffer transient link failures – by decoupling and persistence; (b) dynamic resource availability, processes may leave or join a computation – shared space concept with persistence allows for the data to be stored temporarily at a virtual space; and (c) different administrative domains – by judicious placement of the cache servers on the head nodes, some² of the problems can be overcome.

Our implementation is currently a proof-of-concept system. Although no case studies demonstrating the

²We have not addressed difficulties to achieve communication when firewalls are employed.

actual use of π -channels/ π -Spaces were presented, we discussed how our API may be used in two example applications: a multi-body simulation and a workflow. In the future, we hope to conduct case studies and evaluate the performance impact of the cache server on applications. Also, we have not addressed how to manage a computation when processes join and/or leave. We are considering the use of some coordination language and additional primitives to make this possible.

Acknowledgments

We thank the Victorian Partnership for Advanced Computing (VPAC) for the use of their facilities and for their continuing funding to support this project.

References

- [1] D. Abramson and J. Kommineni. “A Flexible IO Scheme for Grid Workflows”. In *Proc. of the 18th International Parallel and Distributed Processing Symposium*, Krakow, Poland, April 2004.
- [2] L. Bettini and R. D. Nicola. “Mobile Distributed Programming in X-KLAIM”. In M. Bernardo and A. Bogliolo, editors, *Formal Methods for Mobile Computing (SFM-Moby 2005)*, volume 3465 of *Lecture Notes in Computer Science*, pages 29–68. Springer-Verlag, 2005.
- [3] P. Chan and D. Abramson. “NetFiles: A Novel Approach to Parallel Programming of Master/Worker Applications”. In *Proc. of the 5th International Conference and Exhibition on High-Performance Computing in the Asia-Pacific Region (HPCAsia 2001)*, Queensland, Australia, September 2001.
- [4] K. Czajkowski, I. Foster, and C. Kesselman. “Resource Co-Allocation in Computational Grids”. In *Proc. of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 219–228. IEEE Press, August 1999.
- [5] I. T. Foster and K. M. Chandy. “FORTRAN M - A Language for Modular Parallel Programming”. *Journal of Parallel and Distributed Computing*, 26:24–35, 1995.
- [6] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpacesTM Principles, Patterns, and Practice*. Addison-Wesley, 1999.
- [7] D. Gelernter. “Generative communication in Linda”. *ACM Transactions Programming Languages and Systems*, 7(1):80–112, 1985.
- [8] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [9] M. Hsu and A. Silberschatz. “Unilateral Commit: A New Paradigm for Reliable Distributed Transaction Processing”. In *Proc. of the 7th International Conference on Data Engineering*, pages 286–293, Washington, DC, USA, 1991. IEEE Computer Society.
- [10] K. A. Hua, N. Jiang, R. Peng, and M. A. Tantaoui. “PSP: A Persistent Streaming Protocol for Transactional Communications”. In *ICCCAS 2004: Proc. of the 2004 International Conference on Communications, Circuits and Systems*, volume 1, pages 529–533, Washington, DC, USA, 2004. IEEE Computer Society.
- [11] B. K. Johnson and D. J. Ram. “DP: A Paradigm for Anonymous Remote Computation and Communication for Cluster Computing”. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1052–1065, October 2001.
- [12] G. A. Jones and M. Goldsmith. *Programming in OC-CAM 2*. Prentice Hall Professional Technical Reference, 1989.
- [13] D. Kondo, M. Taufer, C. L. B. III, H. Casanova, and A. A. Chien. “Characterizing and Evaluating Desktop Grids: An Empirical Study”. In *Proc. of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS’04)*. IEEE Press, 2004.
- [14] C. Lee. “Stream Programming: In Toto and Core Behavior – Draft Grid Forum White Paper”. [online]. URL: <http://www.eece.unm.edu/~apm/Whitepapers/stream.pdf>. (May 25, 2006), September 2001.
- [15] J. Magee, N. Dulay, and J. Kramer. “Structured Parallel and Distributed Programs”. *Software Engineering Journal*, pages 73–82, March 1996.
- [16] J. Magee, J. Kramer, and M. Sloman. “Constructing Distributed Systems in Conic”. *IEEE Transactions in Software Engineering*, 15(6):663–675, June 1989.
- [17] J. M. Purtilo. “The POLYLITH Software Bus”. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.
- [18] M. V. Reddy, A. V. Srinivas, T. Gopinath, and D. Janakiram. “Vishwa: A Reconfigurable P2P Middleware for Grid Computations”. In *Proc. of the 2006 International Conference on Parallel Processing*. IEEE Press, 2006.
- [19] H. D. Sterck, R. S. Markel, and R. Knight. “A Lightweight, Scalable Grid Computing Framework for Parallel Bioinformatics Applications”. In *HPCS’05: Proc. of the 19th International Symposium on High Performance Computing Systems and Applications*. IEEE Press, 2005.
- [20] H. D. Sterck, R. S. Markel, T. Pohl, and U. Rüede. “A Lightweight Java Taskspaces Framework for Scientific Computing on Computational Grids”. In *SAC2003: Proc. of the ACM Symposium on Applied Computing*, pages 1024–1030, New York, NY, USA, 2003. ACM Press.
- [21] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. “T Spaces”. *IBM Systems Journal*, 37(3):454–474, 1998.
- [22] Y. Zhang and S. Dao. “A ‘Persistent Connection’ Model for Mobile and Distributed Systems”. In *ICCCN ’95: Proc. of the 4th International Conference on Computer Communications*, pages 300–307, Washington, DC, USA, 1995. IEEE Computer Society.