

Persistence and Communication State Transfer in an Asynchronous Pipe Mechanism

Philip Chan, Monash University, Australia
David Abramson, Monash University, Australia*

ABSTRACT

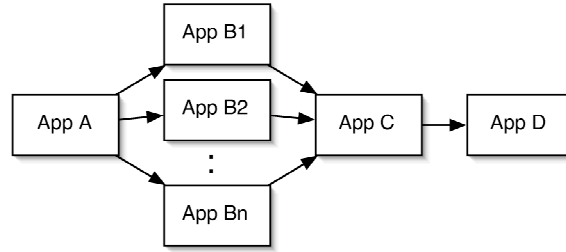
Wide-area distributed systems offer new opportunities for executing large-scale scientific applications. On these systems, communication mechanisms have to deal with dynamic resource availability and the potential for resource and network failures. Connectivity losses can affect the execution of workflow applications, which require reliable data transport between components. We present the design and implementation of π -channels, an asynchronous and fault-tolerant pipe mechanism suitable for coupling workflow components. Fault-tolerant communication is made possible by persistence, through adaptive caching of pipe segments while providing direct data streaming. We present the distributed algorithm for implementing: (a) caching of pipe data segments; (b) asynchronous read operation; and (c) communication state transfer to handle dynamic process joins and leaves. Through communication state transfer, pipe segments from the cache and the writer can be concurrently streamed to the reader. Finally, we present an evaluation of the π -channel implementation.

Keywords: pipe mechanism, asynchronous operation, persistence, fault-tolerant communication, decoupled communication

INTRODUCTION

Heterogeneous distributed systems are the emergent infrastructures for scientific computing. From peer-to-peer, volunteer computing systems to the more structured ensembles of scientific instruments, data repositories, clusters and supercomputers such as computational grids (Foster & Kesselman, 1999), these systems are heterogeneous and dynamic in availability. Furthermore, the wide-area links that interconnect these resources are prone to transient or permanent failures. These dynamic characteristics introduce unique challenges for executing large-scale scientific applications.

Figure 1. A simple four-stage workflow application. Arrows indicate data flow between component applications. Application B is an n-process parallel application.



This research is motivated by the need to support fault-tolerant communication within scientific workflows. A workflow consists of multiple processing stages, where intermediate data generated in one stage are processed in subsequent stages. A workflow component can be a device or an application, which is often modified to enable communication. Thus, a scientific workflow is a computational/data-processing pipeline; with data being captured, processed and manipulated as it pass through various stages (Figure 1). Currently, the data transfers between component applications are realised by: (a) file transfers (e.g. GridFTP); (b) remote procedure calls (e.g. RPC-V, GridRPC, OmniRPC); and (c) custom mechanisms (e.g. Web Services).

For coupling workflow components, we propose the π -channel, an asynchronous and persistent pipe mechanism. It is part of the π -Spaces/ π -channels programming model which features:

1. Simplified application coupling using string channel names through π -Spaces. A π -Space is a name space for π -channels, enabling dynamic binding of channel endpoints between processes.
2. π -channel data are adaptively cached to achieve persistence. This allows π -channels to be created and written to, even in the absence of the reader. Persistence also makes π -channels accessible even after the writer has terminated.
3. Asynchronous receives are made possible through a communication thread; thus, an application is able to accept pipe segments even when it is busy in computation.

This article focuses on how π -Space persistence relates to fault-tolerant communication in scientific workflows. The extended API and semantics for π -Space/ π -channels are presented. We describe the design and implementation of π -channels, including the server that implements this model along with the underlying distributed algorithm.

This article is organised as follows: We review related work in the next Section § 2. Then, we present the π -Spaces/ π -channels programming model in § 3, including its application programming interface, semantics, and how fault-tolerance is achieved for workflows. In § 4, we discuss in detail its design and implementation, describing the distributed algorithm. Experimental results are presented in § 5, followed by the conclusions.

RELATED WORK

We briefly review the major models for communication on distributed environments highlighting their differences from π -Spaces/ π -channels.

Pipe/Channel Models

The pipe/channel is a well-known IPC mechanism and appears in many forms: Unix pipes, named pipes, and TCP sockets (Stevens, 1998). Sockets with TCP, while used in network programming, are too low-level for scientific application programming. In particular, since communication endpoints are identified using IP/host addresses and port numbers, it is tedious to

use in a dynamic, failure-prone environment. In the event of a link failure, TCP primitives will generate a “broken pipe” exception, which require explicit handling.

The channel abstraction has its early beginnings in Kahn’s Process Networks (Kahn, 1974) and Communicating Sequential Processes (Hoare, 1985). This abstraction is the basis of current process calculi. Many coordination languages use the channel model, e.g. MoCha (Guillen-Scholten & Arbab, 2005), POLYLITH (Purtilo, 1994), Programmer’s Playground (Goldman et al., 1995), Conic (Magee et al., 1989), and Netfiles (Chan & Abramson, 2001). Channels are provided in Grid programming environments, e.g. Ibis (van Nieuwpoort et al., 2005) and Vishwa (Reddy et al., 2006). Vishwa applications communicate through a pipe mechanism called DP (Johnson & Ram, 2001). However, these systems do not support fault-tolerance in the communication.

Communication persistence was previously explored by at least the following: (a) persistent pipes for transactions (Hsu & Silberschatz, 1991); (b) persistent connections over TCP (Zhang & Dao, 1995); and more recently, (c) the persistent streaming protocol (Hua et al., 2004) and (d) NapletSocket (Zhong & Xu, 2004). Unlike π -Spaces/ π -channels, these lack a logical name space for communication endpoint coupling. For example, IP addresses and ports are necessary for configuring endpoints in (Hsu & Silberschatz, 1991) and (Zhang & Dao, 1995). Moreover, π -Spaces/ π -channels support asynchronous communication.

Message-oriented middleware (MoM), e.g. IBM’s Websphere MQ (IBM Websphere MQ, 2008) and the Microsoft Message Queueing System (MSMQ) (Microsoft, 2008), present a suite of asynchronous communication services suitable for general transaction processing. The message transfer times are in the order of minutes instead of seconds or milliseconds (Tanenbaum & Steen, 2007), reducing their applicability for high-performance scientific applications. Furthermore, queue management requires tedious setup and configuration, while the π -Spaces/ π -channels model is designed for efficient pipe creation/retrieval, including dynamic binding of channel endpoints.

π -channels in Context with Netfiles and GriddLeS

Our earlier work on Netfiles (Chan & Abramson, 2001, 2008) investigated file I/O as metaphor for interprocess communication. The idea was inspired from the Nimrod project (Abramson et al., 1995, 1997). Nimrod is a middleware for the executing large-scale parametric models (or sweeps) over distributed systems. These parametric sweep applications can be built without modifying the existing programs. For many such applications, the component programs communicate by through data files, with file transfers performed by the runtime.

In GriddLeS (Abramson & Kommineni, 2004), the file I/O metaphor is extended for wide-area environments like Grids, implemented over Web Services. The GriddLeS runtime provides an I/O multiplexer, which transparently performs file transfers and buffered remote I/O operations to couple applications that read/write files. This enables Grid workflows to be composed without rewriting any program code, a feature useful when existing legacy codes are executed over computational grids.

The π -channel abstraction extends Netfiles and GriddLeS with persistence and efficient asynchronous operations. Furthermore, GriddLeS offer static associations of names to process locations, while π -Spaces/ π -channels provide dynamic π -channels binding.

Message Passing

Message Passing Interface (MPI) is widely used for writing parallel programs. However, Grid applications like scientific workflows require coupling of multiple separate applications. For this, the MPI-2 standard provides Unix socket-like interface for accepting and establishing connections between two MPI applications, enabling communication with `MPI_Send()` and `MPI_Recv()`. Although fault-tolerance may be incorporated into this mechanism, current projects (Fagg &

Dongarra, 2004; Batchu et al., 2004; Bouteiller et al., 2006; Gropp & Lusk, 2004) are focused on fault-tolerant IPC within an application.

In the MPI model, each process is identified by an integer rank. Elegant and simple, this model works very well on SPMD applications where the number of processes is known and fixed. Phoenix (Taura et al., 2003), for example, modifies the process naming scheme so that processes may join and leave the computation without the need to re-assign ranks. When coupling multiple applications in a workflow, it is useful to have a user-intuitive convention to identify communication endpoints (Chan & Abramson, 2007, p. 6).

Generative Communication Models

Linda (Carriero & Gelernter, 1989) is a generative communication model that features decoupled communication. In Linda, processes communicate by posting and retrieving ordered sequences of values called tuples onto a logical shared space called tuplespace. Its elegance has inspired many systems such as Sun's Javaspaces (Freeman et al., 1999), and IBM's T-spaces (Wyckoff et al., 1998). The Linda tuplespace model encourages decoupled communication along two dimensions. First, tuples are posted and retrieved anonymously from tuplespace, achieving space decoupling. Second, since tuples are persistent, temporal decoupling is possible, allowing non-concurrent processes to communicate.

Workflows require efficient data transfers, which is challenging to achieve in Linda. Extensions have thus been proposed, e.g. WCL (Rowstron, 1998) supports bulk transport of tuples. Taskspaces (Sterck et al., 2003, 2005) provide direct communication using the tuples to identify communication endpoints (IP/port) and TCP to connect processes. However, it lacks support for communication fault-tolerance.

π -CHANNELS: A PERSISTENT PIPE MECHANISM

A π -channel is an enhanced unidirectional (MRSW) pipe that has a unique user-specified string name and a FIFO sequence of arbitrary-length typed data segments, each treated as indivisible units. Fault-tolerant communication is achieved by persistence, enabling π -channels to be created and written to at any time, independent of the sink/reader, thus encouraging temporal decoupling.

During π -channel creation, if the matching reader is known, a direct connection (if possible) is used to efficiently transfer pipe segments. Due to persistence (Chan & Abramson, 2007), delivery of the π -channel to the π -Space continues (Figure 2a). This enables π -channel writes to proceed even if the link and/or reader have failed during communication. At this point, the pipe segments will be written to the π -Space (Figure 2b). When the reader resumes, a communication state transfer re-establishes the connections between the reader, the π -Space, and the writer.

Figure 2. Dual π -channel behaviours: (a) when reader-to-writer link is available; and (b) during a link failure, writer continues transmitting data to the space, delivery to the reader resumes upon link restoration.

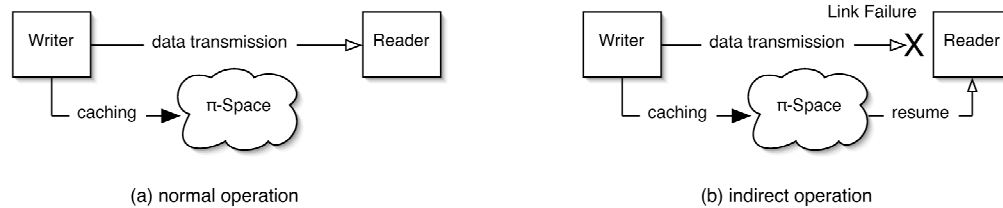


Table 1: A brief overview of the key π -channels primitives.

π -channel Primitive	Brief Description
<code>pi_create()</code>	Creates a new π -channel.
<code>pi_attach()</code>	Attach and retrieve a given π -channel.
<code>pi_write()</code>	Writes a pipe segment.
<code>pi_read()</code>	Reads a pipe segment.
<code>pi_seek()</code>	Seek to a new read segment position.
<code>pi_tell()</code>	Return the segment ID of upcoming segment.
<code>pi_close()</code>	Closes a π -channel that is opened for writing.
<code>pi_detach()</code>	Detach from reading a π -channel.
<code>pi_unlink()</code>	Marks a closed p-channel for deletion.

π -channels: Programming Model and Semantics

Table 1 summarises the API for π -channels. This is inspired from file-based I/O and connection-oriented socket communication. The key difference is that a thread is employed to receive data asynchronously. The read/write operations resemble the standard Unix I/O operations, with specifications of the segment data type, for heterogeneous communication.

The π -channel API

The π -channels programming interface consists of the following:

1. `pi_attach(str n, int s)`
attaches to a π -channel with name `n` on space `s`, does not block even if π -channel is absent. It initiates asynchronous receive, allowing segments to be buffered locally even before the `pi_read()` is issued. This primitive returns a descriptor representing the “read” end.
2. `int pi_create(str n, int s, int mode)`
creates a π -channel with name `n` on space `s`. The `mode` argument is used to specify if this is a `CREATE` – for new π -channels; or `APPEND` – to resume writing. On success, it returns a descriptor representing the “write” end of the created π -channel.
3. `int pi_read(int d, ptr b, int len, dtype_t1 t)`
reads a segment of `len` elements of type `t` into buffer `b` from the descriptor `d`, blocks if no segments are available. It returns the number of elements successfully read or `-1` if end of the pipe is reached.
4. `int pi_write(int d, ptr b, int len, dtype_t t)`
writes a segment of `len` elements of type `t` from buffer `b` into the descriptor `d`.
5. `int pi_close(int d), int pi_detach(int d)`
closes the “write” and “read” end of the π -channel `d`, respectively.
6. `int pi_seek(int d, int seg_id)`
moves the logical pipe pointer for `d` to start reading from segment `seg_id`, with `0` as the first segment. This only works at the read end. On success, it returns `0`.
7. `pi_tell(int d)`
returns the segment ID of the logical pipe pointer of the π -channel.
8. `pi_unlink(str n)`
marks the π -channel `n` for deletion, returns `-1` for open π -channels.

¹The `dtype_t` covers various data types, e.g. `PI_INT`, `PI_FLOAT`, etc.

With persistence, a π -channel behaves as both: (a) an archival file – writes can proceed without readers; and (b) an online pipe – when the reader/s and the writer are concurrent. This duality makes `pi_seek()` possible, when such an operation would be meaningless on conventional pipes. All π -channels are immutable, so `pi_seek()` is disabled at the “write” end.

Semantics of π -channel Operations

Figure 3. The states of a π -channel.

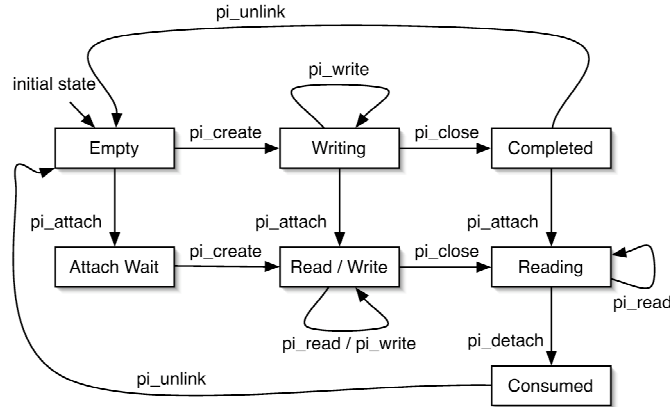


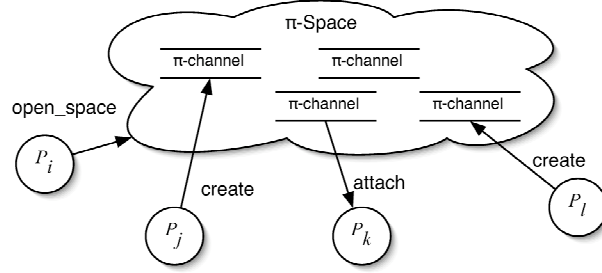
Figure 3 shows a state-transition diagram for π -channels. The top three states show the lifecycle of a π -channel as it is created, written-to, closed, and deleted. The remaining states show the transitions when there is a concurrent reader. In particular, the Read/Write applies when the π -channel has one active writer and at least one active reader. At the Reading state, the π -channel behaves like a stored file object, with data segments retrieved from π -Space.

This simple semantics facilitates a straightforward failure recovery mechanism. The key is to enable the application to resume π -channel operations upon recovery, without affecting its correctness. During writer recovery, `pi_create()` and `pi_write()` operations are “redone.” If segments are already cached, the sequence of `pi_write()` operations are replayed, without changing π -channel segments that are already on π -Space. Eventually, a new segment is written marking to start of normal operation. During reader recovery, all `pi_attach()` and `pi_read()` are performed on cached π -channels, with pipe segments delivered from the cache. Processes are assumed to be piecewise deterministic, thus able to repeat the results it generated in a previous failed execution.

π -Spaces: Abstraction of Shared Space for π -Channels

Processes coordination is through one or more π -Spaces. Similar to Linda tuplespace, a π -Space is a shared space abstraction for π -channels, which are explicitly posted and retrieved by processes (Figure 4). A π -Space is a logical name space for π -channels, providing dynamic binding of channel endpoints.

Figure 4. Shared space communication in π -Space/ π -channels.



Persistence fits elegantly with the model of shared spaces. When a π -channel is created, a copy is automatically posted on the π -Space. During a `pi_attach()`, the π -Space is accessed to retrieve the named π -channel. Within a single π -Space, a name is bound to at most one π -channel. To reuse an existing name, the π -channel has to be marked for deletion. The programming model includes operations to create, access, and close π -Spaces, enabling the use of multiple spaces within a single application.

Support for Fault-Tolerance and Application Migration

During a workflow execution, the following events may occur:

1. The source application (writer) leaves the workflow. Unless the downstream components do not require any further data from this writer, the entire workflow may be stalled. This also occurs when all outstanding data segments has been consumed.
2. The sink application (reader) leaves the workflow. The writer continues streaming to the cache (π -Space), ignoring the loss of the reader. When the reader recovers, it can resume reading from the π -Space.
3. The link between applications is severed. Assuming that the π -Space is implemented as a reliable service; this is considered as a combination of source failure – from the perspective of the sink; and sink failure – from the perspective of the source.

An application may leave a workflow voluntarily or involuntarily. A voluntary departure occurs when it migrates to another resource. Communication state transfer is employed to re-establish connections with migrated applications. An involuntary departure may be due to machine failure and/or application crash. We assume the fail-stop failure model, i.e., the process crashes and performs no further communication. A connection loss is treated as a component failure and is detected when a communication operation cannot be completed.

DESIGN AND IMPLEMENTATION

The π -Spaces/ π -channels model is implemented as an API with a runtime system, and a multi-threaded server that provides the dynamic lookup and storage of π -channels. A π -Server manages a single π -Space, which encourages deploying multiple servers to improve the distribution of load. Furthermore, a π -Server may execute at the cluster head node, serving as a communication gateway to processes on remote clusters.

Basic Components and General Functional Description

Figure 5. Client-side components and design of the π -Server.

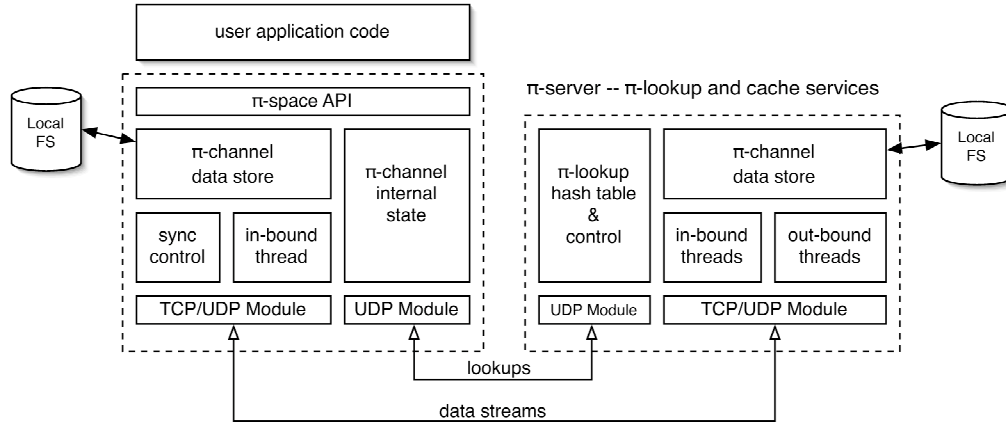


Figure 5 (right) presents the π -Server components. The server maintains a thread pool for incoming and outgoing transmissions, providing a non-archival pipe storage service. The thread count is configurable to support different application loads. The in-bound threads handle incoming data streams from sources, while out-bound threads forward/push π -channel data to sinks. Data streams are transported using a TCP protocol, while lookups are implemented using a lightweight UDP protocol with retransmission capability.

Figure 5 (left) shows the client-side components. Each π -Spaces/ π -channels application is capable of accepting incoming transmissions asynchronously. During a `pi_attach()`, the reader informs the server of its identity. This allows data streams to be forwarded to the reader while it is busy performing computations. A single event-driven in-bound thread stores incoming segments into a buffer (by the data store component). When buffer capacity is reached, data segments are stored into a local file, identified by the `pipe_id`. During a `pi_read()`, this buffer/file is accessed to retrieve the requested segment.

General Description of the Distributed Algorithm

Figure 6 presents the distributed algorithm for the π -Spaces/ π -channels. We adopt the Python convention to indicate block structure (i.e., the statement alignment determines a block). The notation (`#line_no`) is used in-text when referring to the algorithm.

The pair (`pipe_id`, `space_id`) represents a system-wide π -channel identifier, and `pipe_id` is unique within each π -Space. Since the algorithm assumes a single π -Space, we remove reference to the `space_id`.

Figure 6. Distributed algorithm for π -Spaces/ π -channels.

Client API

```

1  pi_attach ( n ):
2    send ( get, n, my_id ) to  $\pi$ -Server
3    recv ( get, &pipe_id ) from  $\pi$ -Server
4    id  $\leftarrow$  free entry on local id_tab
5    if buffer for pipe_id does not exist:
6      create buffer for pipe_id
7    associate pipe_id & buffer with id
8    return id

9  pi_create ( n ):
10   send ( put, n, my_id ) to  $\pi$ -Server
11   recv ( get, &pipe_id, &dest_list ) from  $\pi$ -Server
12   foreach dest in dest_list:
13     fd_list [dest]  $\leftarrow$  connect to dest
14   id  $\leftarrow$  free entry on local id_tab
15   associate fd_list & pipe_id with id
16   return id

17 pi_read ( id, m, len ):
18   off  $\leftarrow$  compute offset
19   buffer  $\leftarrow$  retrieve buffer for id
20   block until seg [ off, len ] in buffer || eoc
21   if segment was found:
22     get seg [ off, len ] from buffer
23     store segment into m
24     return len
25   return -1 /*end of channel */

26 pi_write ( id, m, len ):
27   pipe_id  $\leftarrow$  map pipe_id from id
28   fd_list  $\leftarrow$  retrieve fd_list for id
29   off  $\leftarrow$  compute offset
30   success_count  $\leftarrow$  0
31   foreach dest in fd_list:
32     write SEG ( pipe_id, m, off, len ) to dest
33     if write successful:
34       success_count++
35   if success_count < len(fd_list):
36     check_restore(pipe_id)
37   update status of pipe_id
38   return len

39 check_restore ( pipe_id ):
40   query  $\pi$ -Server for status of migrated reader
41   if migrated reader found:
42     fd  $\leftarrow$  connect to migrated reader
43     update fd_list with new fd
44     remove reader from hold_list

```

Algorithm of the π -Server

```

45 On ( get, &n, &src ):
46   entry  $\leftarrow$  find n on  $\pi$ -table
47   if entry does not exist:
48     entry  $\leftarrow$  create n on  $\pi$ -table
49   entry.reader  $\leftarrow$  src
50   entry.has_reader  $\leftarrow$  true
51   pipe_id  $\leftarrow$  entry.pipe_id
52   send ( get, pipe_id ) to src
53   if entry.is_cached:
54     initiate forwarding to src

55 On ( put, &n, &src ):
56   entry  $\leftarrow$  find n on  $\pi$ -table
57   if entry does not exist:
58     entry  $\leftarrow$  create n on  $\pi$ -table
59   dest_list  $\leftarrow$  ( )
60   if entry.has_reader:
61     append entry.reader to dest_list
62   append my_id to dest_list
63   pipe_id  $\leftarrow$  entry.pipe_id
64   entry.is_cached  $\leftarrow$  true
65   send ( put, pipe_id, dest_list ) to src

```

Algorithm of the in-bound thread

```

66 On CON ( &src, &pipe_id ):
67   fd  $\leftarrow$  accept inbound connection
68   buffer  $\leftarrow$  retrieve buffer for pipe_id
69   if buffer does not exist:
70     create buffer for pipe_id
71   associate fd with pipe_id

72 On SEG ( &pipe_id, &m, &off, &len ):
73   buffer  $\leftarrow$  retrieve buffer for pipe_id
74   store m at offset off

```

Algorithm of the out-bound thread

```

75 if forwarding  $\pi$ -channel pipe_id to dest:
76   fd  $\leftarrow$  open connection to dest
77   foreach segment  $\in$  local buffer:
78     send segment to fd

```

Each participating application maintains the following variables:

1. `my_id` – unique ID, implemented as an IP/port pair. The port number is dynamically generated during application startup.
2. `id_tab` – a local table (#4, #14) associating the `pipe_id` with open file descriptors `fd_list`, channel read-write pointers, and other local state information. This table corresponds to the π -channel internal state component in Figure 5.

During a `pi_create()`, a put request (#10) is sent to the π -Server, which creates (#58–#65) an entry for this π -channel on a hash table and returns a unique `pipe_id`. It replies (#58, #61, #62) with a list (possibly empty) of destinations. If the reader’s identity is known, the reader’s address appears first, followed by the server’s address. The `pi_create()` establishes (#12, #13) a connection with the destinations and associates the pipe with the file descriptors. It returns as descriptor (#14, #16) the position of the π -channel on `id_tab`.

The `pi_attach()` sends a get request to the server (#2), which replies (#51, #52) with the unique `pipe_id` for the π -channel, even if non-existent. The server creates an entry for this π -channel, storing the reader’s address for use in channel creation.

A `pi_read()` does not read directly from the open connection with the source. Instead, incoming data segments are handled by the in-bound thread (#66–#74), which listens and accepts TCP operations on behalf of the application. The π -Server manages a thread pool for the same purpose of enabling asynchronous read operations. When the in-bound thread accepts a π -channel, it allocates a buffer (#70) for pipe segments. Each segment (#72–#74) contains type information, length, offset, and `pipe_id`. The received segments are stored in a shared buffer, so that `pi_read()` can retrieve (#19, #22) them. The out-bound thread pushes π -channels to sinks (#54, #75–#78). During reader recovery, these threads send missed segments to the reader.

We only outline the migration mechanism (Figure 7), showing when application state is saved and restored after migration. The idea (#79–#86) is to attempt a graceful connection shutdown before migrating. Since pipes are cached, undelivered data segments can be retrieved from the π -Server. The `hold_list` (#89, #44) identifies the migrating processes.

Communication state migration, similar to (Chanchio & Sun, 2004), performs a connection hand-over with the migrated reader (#39–#43). In Figure 8, the migrated peer re-establishes connection with the writer so that: (1) Seg 2 is retrieved from the π -Server; and at the same time (2) Seg 3 is streamed from the writer.

Figure 7. The communication migration protocol.

```

79  if I am migrating:
80      disable sending acks for heartbeats
81      migrating ← true
82      foreach open  $\pi$ -channel:
83          save offset into checkpoint
84          flush and close all connections
85          perform local state checkpoint
86          send checkpoint to the new location

87  if a peer is migrating:
88      /* reject connections from this list */
89      add peer_addr to hold_list

```

Figure 8. Time diagrams showing concurrent reading of a π -channel from π -Server and writer. In (a), the migrated reader resumes reading from cache. In (b), it also resumes connection with the writer.

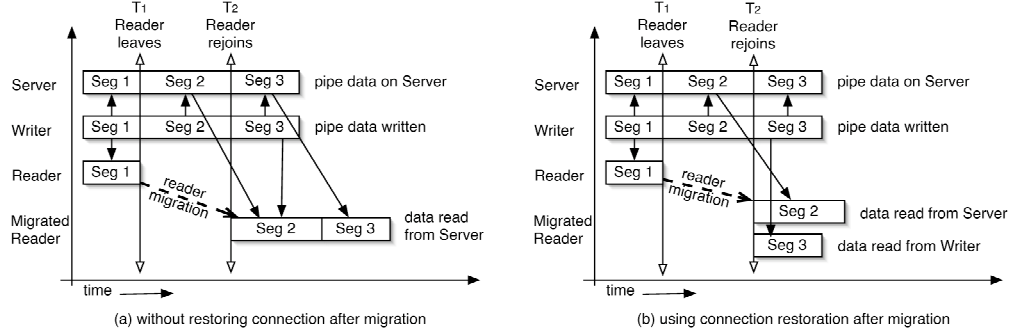


Table 2: Participating systems in our experiments.

Name	Location	Processor	OS	#CPUs
mahar.infotech.monash.edu.au	Monash	Intel P4	Linux 2.4.27-3	50
edda.vpac.org	VPAC	IBM Power5	SLES 9 Linux	80
wexstan.vpac.org	VPAC	AMD Opteron	Red Hat Linux	246
tango.vpac.org	VPAC	AMD Opteron	CentOS 5 Linux	760

Table 3. UDP request-reply performance within a cluster.

Number of Clients	2	4	8	16	32
Number of Requests Served	80 000	160 000	320 000	640 000	1 280 000
Mean Execution Times (s)	21.99	25.51	38.72	44.65	96.58
Standard Deviation	0.822	0.813	0.741	0.430	0.170
Request Rate (per second)	3 642.6	6 277.6	8 267.1	14 335	13 390
Standard Deviation	138.89	200.85	162.38	137.58	23.82

EXPERIMENTAL RESULTS

Two aspects of the implementation are evaluated. First, we measure the rate in which π -channel lookup operations are handled by the π -Server under two scenarios: (a) π -Server and clients are on one cluster; and (b) clients perform lookups over a wide-area network. Second, we measure the throughput when communication takes place between two applications over our WAN testbed. This test shows how asynchronous read operations improve the bandwidth utilisation. Table 2 lists the resources we used. VPAC (Victorian Partnership for Advanced Computing) is an HPC consortium of universities in Victoria, Australia. Our wide-area testbed uses both Monash and VPAC resources.

π -Server Lookup Performance

We evaluate the request-handling rate of the π -Server, with up to 32 clients concurrently generating lookup requests. Table 3 presents results conducted over mahar, measuring the execution time of all clients when looking up 25 366 unique but randomly generated π -channel names. Each client performed 40 000 lookups, without channel read/write operation. Clients were assigned on execute nodes, with the π -Server on the head node. At least 12 runs were performed for each test case, using only the timings from the middle 10 runs.

Table 4. UDP request-reply performance on a WAN.

Number of Clients	2	4	8	16	32
Number of Requests Served	10 000	20 000	40 000	80 000	160 000
Mean Execution Times (s)	10.58	11.02	11.52	12.34	24.74
Standard Deviation	0.518	0.340	0.285	0.122	0.268
Request Rate (per second)	946.4	1 815.6	3 473.8	6 480.8	6 467.6
Standard Deviation	46.32	56.53	86.53	64.07	70.39

On Table 4, we present the timings for lookups on a WAN between Monash and VPAC. Clients ran on mahar compute nodes with the π -Server running on wexstan’s head node, using the same parameters as in the LAN tests. These results indicate that the bottleneck for grid applications will most likely be the high latencies between the π -Server and the clients.

π -channel Throughput on a Multi-Cluster Testbed

We evaluate and compare the data transfer throughput using π -channels under three scenarios: (a) matched create/attach operations; (b) create first, then attach; and (c) asynchronous read operations. Two processes are executed, using a pair of π -channels for communication.

For scenario one, processes use paired `pi_attach()` and `pi_create()` operations, i.e., when one process is writing, the other is reading. For the second scenario, each process executes all π -channel writes first, followed by reads. The π -Server caches most of the pipe segments during the write phase. Once the processes perform the `pi_attach()`, the pipe segments are retrieved from the π -Server rather than the writer.

In the third scenario, processes initiate a non-blocking `pi_attach()` on an inbound channel first, before a `pi_create()`, followed by the write and then read operations. This notifies the π -Server of a pending request for a channel, providing writers with the destination addresses and encouraging direct streaming. The performance results show a reduced dependence on the π -Server to store pipe segments.

Figure 9 shows the segment send rates while Figure 10 presents the measured bandwidth. The π -Server executed on edda, one application on tango and another on wexstan. The best performance is achieved under asynchronous operation (scenario three). The detailed results are presented on Table 5. The tests were conducted with at least 12 trials per case. Of these, the mean is computed using ten results, discarding the highest and lowest values. The standard deviation given is for the throughput, i.e., the mean message send rates for each of the segment sizes. Note the absence of observable performance differences for Scenario 1 and 2. This means that matching `pi_attach()` with `pi_create()` operations does not lead to any improvement in data transfer rates. Thus, π -Space/ π -channels applications may be written without using an odd-even rule to match reads and writes.

In Scenario 3, the use of `pi_attach()` notifies the π -Server of a pending request to retrieve a π -channel. This makes it possible for the writer of that π -channel to transmit the channel segments directly to the reader, showing a substantial improvement in the data transfer performance over that of the first two scenarios. Furthermore, the overlap of sends and receives results in better utilisation of the available bandwidth.

Figure 9. Measured message-send rates with a ping-pong benchmark.

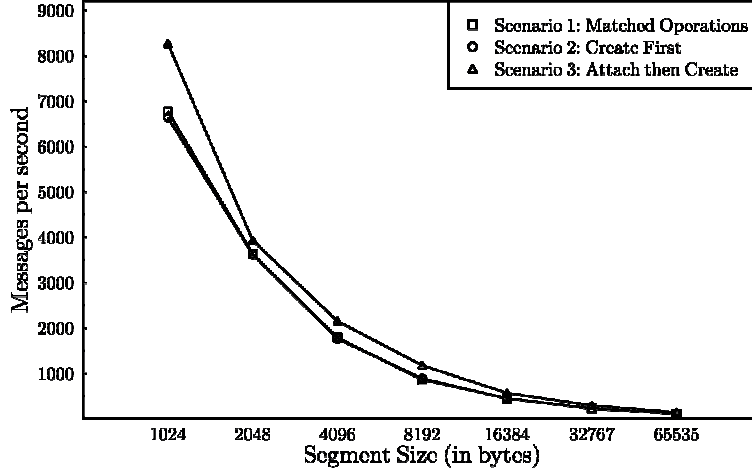
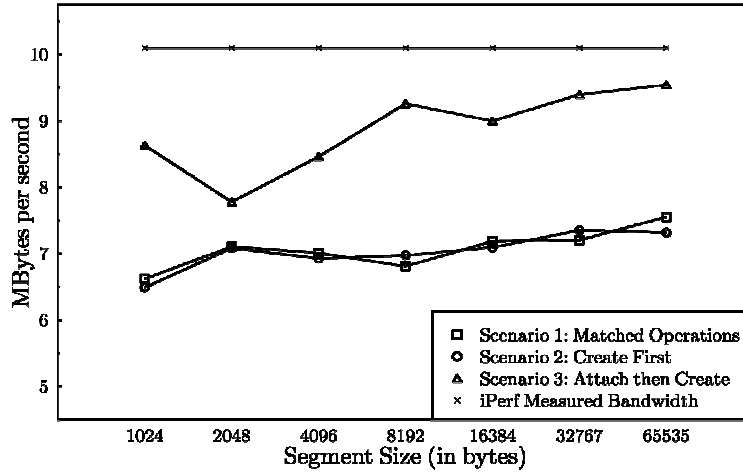


Figure 10. Measured bandwidth with a ping-pong π -Space/ π -channels application. The horizontal bar shows the measured bandwidth using *iperf*².



CONCLUSIONS

We have presented π -Spaces/ π -channels, a communication mechanism for scientific workflows on dynamic environments, where resources may fail and network links may be disrupted. The key feature of π -channels is persistence, enabling communication despite process failures or departures. This article presents its design and implementation. In particular, we describe the distributed algorithm showing how persistence is achieved with the caching mechanism and the asynchronous operation using the in-bound thread. A communication state transfer mechanism is employed, which re-establishes connections with migrated components to improve the data stream transfer time. Experimental results show that the caching mechanism is able to buffer channel data

² <http://sourceforge.net/projects/iperf>

Table 5. Results from the bandwidth and throughput measurements.

Size of Segment (bytes)	1024	2048	4096	8192	16384	32767	65535
Scenario 1							
Throughput	6781.586	3638.966	1794.872	871.818	460.053	230.603	120.902
Std. Dev.	186.270	25.994	37.553	7.248	7.471	3.927	1.467
Bandwidth (MB/s)	6.623	7.107	7.011	6.811	7.188	7.206	7.556
Scenario 2							
Throughput	6646.947	3627.420	1773.944	892.972	454.091	235.335	117.091
Std. Dev.	140.815	26.239	29.443	7.959	7.517	2.163	1.622
Bandwidth (MB/s)	6.491	7.085	6.929	6.976	7.095	7.354	7.318
Scenario 3							
Throughput	8841.626	3983.438	2166.548	1185.448	575.914	300.792	152.712
Std. Dev.	249.939	15.483	19.652	14.235	8.900	4.880	1.309
Bandwidth (MB/s)	8.634	7.780	8.463	9.261	8.999	9.399	9.544

segments and when asynchronous operation is employed, throughput is substantially increased. With asynchronous operation, sending and receiving of messages can be overlapped, resulting in improved message sending rates than normal non-asynchronous operation.

ACKNOWLEDGMENTS

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

REFERENCES

- Abramson, D., Foster, I., Giddy, J., Lewis, A., Sasic, R., Sutherst, R., & White, N. (1997). "Nimrod Computational Workbench: A Case Study in Desktop Metacomputing". In *Australian Computer Science Conference (ACSC 97)*. Macquarie University, Sydney.
- Abramson, D. & Kommineni, J. (2004). "A Flexible IO Scheme for Grid Workflows". In *Proc. of the 18th International Parallel and Distributed Processing Symposium*, Krakow, Poland.
- Abramson, D., Sasic, R., Giddy, J., & Hall, B. (1995). "Nimrod: A Tool for Performing Parameterised Simulations using Distributed Workstations". In *Proc. of the 4th IEEE Symposium on High Performance Distributed Computing*, Virginia. IEEE Press.
- Batchu, R., Dandass, Y. S., Skjellum, A., & Beddhu, M. (2004). "MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware ". *Cluster Computing*, 7(4): 303–315.
- Bouteiller, A., Hérault, T., Krawezik, G., Lemarinier, P., & Cappello, F. (2006). "MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI". *International Journal of High-Performance Computing Applications*, 20(3): 319–333.
- Carriero, N. & Gelernter, D. (1989). "How to Write Parallel Programs: A Guide to the Perplexed". *ACM Computing Surveys*, 21(3): 323–357.
- Chan, P. & Abramson, D. (2001). "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.
- Chan, P. & Abramson, D. (2007). "π-spaces: Support for Decoupled Communication in Wide-Area Parallel Applications". In *Proc. of the Sixth International Conference on Grid and Cooperative Computing*, pages 3–10, Urumchi, Xinjiang, China. IEEE.
- Chan, P. & Abramson, D. (2008). "Netfiles: An Enhanced Stream-based Communication Mechanism". In Labarta, J., Joe, K., and Sato, T., editors, *High-Performance Computing, Revised Selected Papers. Sixth International Symposium, ISHPC 2005 and First International Workshop on Advanced Low Power Systems, ALPS 2006*, volume 4759 of *Lecture Notes in Computer Science*, pages 254–261. Springer-Verlag.

- Chanchio, K. & Sun, X.-H. (2004). "Communication State Transfer for Mobility of Concurrent Heterogeneous Computing". *IEEE Transactions on Computers*, 53(10):1260–1273.
- Fagg, G. E. & Dongarra, J. (2004). "Building and Using a Fault-Tolerant MPI Implementation". *International Journal of High-Performance Computing Applications*, 18(3):353–361.
- Foster, I. & Kesselman, C. (1999). "Computational Grids". In *The Grid: Blueprint for a New Computing Infrastructure*, pages 15–51.
- Freeman, E., Hupfer, S., & Arnold, K. (1999). *JavaSpacesTM Principles, Patterns, and Practice*. Addison-Wesley.
- Goldman, K. J., Swaminathan, B., McCartney, T. P., Anderson, M. D., & Sethuraman, R. (1995). "The Programmers' Playground: I/O Abstraction for User-Configurable Distributed Applications". *IEEE Transactions in Software Engineering*, 21(9):735–746.
- Gropp, W. & Lusk, E. (2004). "Fault Tolerance in Message Passing Interface Programs". *International Journal of High-Performance Computing Applications*, 18(3):363–372.
- Guillen-Scholten, J. & Arbab, F. (2005). "Coordinated Anonymous Peer-to-Peer Connections with MoCha". In Guelfi, N., Reggio, G., and Romanovsky, A., editors, *Scientific Engineering of Distributed Java Applications, Revised Selected Papers. 4th International Workshop, FIDJI 2004*, volume 3409 of *Lecture Notes in Computer Science*, pages 68–77. Springer-Verlag.
- Hoare, C. (1985). *Communicating Sequential Processes*. Prentice Hall.
- Hsu, M. & Silberschatz, A. (1991). "Unilateral Commit: A New Paradigm for Reliable Distributed Transaction Processing". In *Proc. of the 7th International Conference on Data Engineering*, pages 286–293. IEEE Computer Society.
- Hua, K. A., Jiang, N., Peng, R., & Tantaoui, M. A. (2004). "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. IEEE Computer Society.
- IBM Websphere MQ (2008). The IBM Websphere MQ Family. [online]. URL: <http://www.ibm.com/software/websphere>. (March, 2008).
- Johnson, B. K. & Ram, D. J. (2001). "DP: A Paradigm for Anonymous Remote Computation and Communication for Cluster Computing". *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1052–1065.
- Kahn, G. (1974). "The Semantics of Simple Language for Parallel Programming". In *Proc. of the 1974 IFIP Congress*, pages 471–475.
- Magee, J., Kramer, J., & Sloman, M. (1989). "Constructing Distributed Systems in Conic". *IEEE Transactions in Software Engineering*, 15(6):663–675.
- Microsoft (2008). Microsoft Message Queueing. [online]. URL: <http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx> (March, 2008).
- Purtilo, J. M. (1994). "The POLYLITH Software Bus". *ACM Transactions on Programming Languages and Systems*, 16(1):151–174.
- Reddy, M. V., Srinivas, A. V., Gopinath, T., & Janakiram, D. (2006). "Vishwa: A Reconfigurable P2P Middleware for Grid Computations". In *Proc. of the 2006 International Conference on Parallel Processing (ICPP 2006)*. IEEE Press.
- Rowstron, A. (1998). "WCL: A Co-ordination Language for Geographically Distributed Agents". *World Wide Web*, 1(3):167–179.
- Sterck, H. D., Markel, R. S., & Knight, R. (2005). "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.
- Sterck, H. D., Markel, R. S., Pohl, T., & Rüede, U. (2003). "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. ACM Press.
- Stevens, W. R. (1998). *Unix Network Programming: Networking APIs: Sockets and XTI*, volume 1. Prentice-Hall PTR, 2nd edition.
- Tanenbaum, A. S. & Steen, M. V. (2007). *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2 edition.
- Taura, K., Kaneda, K., Endo, T., & Yonezawa, A. (2003). “Phoenix: A Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources”. In *PPoPP '03: Proc. of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 216–229, New York, NY, USA. ACM Press.
- van Nieuwpoort, R. V., Maassen, J., Wrzesinska, G., Hofman, R. F. H., Jacobs, C. J. H., Kielmann, T., & Bal, H. E. (2005). “Ibis: a Flexible and Efficient Java-based Grid Programming Environment”. *Concurrency and Computation: Practice and Experience*, 17(7–8):1079–1107.
- Wyckoff, P., McLaughry, S. W., Lehman, T. J., & Ford, D. A. (1998). “T Spaces”. *IBM Systems Journal*, 37(3):454–474.
- Zhang, Y. & Dao, S. (1995). “A ‘Persistent Connection’ Model for Mobile and Distributed Systems”. In *ICCCN '95: Proc. of the 4th International Conference on Computer Communications*, pages 300–307. IEEE Computer Society.
- Zhong, X. & Xu, C.-Z. (2004). “A Reliable Connection Migration Mechanism for Synchronous Transient Communication in Mobile Codes”. In *Proc. of the 2004 International Conference on Parallel Processing*. IEEE Press.