

Persistence and Communication State Transfer in an Asynchronous Pipe Mechanism

Philip Chan

David Abramson

*MESSAGE Laboratory**Caulfield School of Information Technology, Monash University
900 Dandenong Road, Caulfield East, Victoria, Australia*

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

Abstract—Emergent wide-area distributed systems like computational grids present opportunities for large scientific applications. On these systems, communication mechanisms have to deal with dynamic resource availability and occurrence of network failures. In this paper, we present the design and implementation of an asynchronous and persistent pipe mechanism, called π -channels. These communication issues are addressed by combining adaptive caching with data streaming for efficient and fault-tolerant communication. We present the underlying distributed algorithm that implements (a) caching of pipe data segments; (b) asynchronous operation; and (c) re-establishment of connections when a peer leaves and rejoins the computation – part of a communication state transfer mechanism. This makes it possible for different segments (from cache and from writer) of the pipe data to be concurrently streamed to the migrated reader, reducing the retrieval time. Finally, we present some performance results showing the benefits of asynchronous operation.

I. INTRODUCTION

Wide-area distributed systems like computational grids have become the emergent infrastructures for scientific computing. Advances in technology have made it easier to aggregate diverse computing resources spread over a wide-geographic area for solving large scientific problems. The current trend is to employ: (a) peer-to-peer computing; and (b) computational grids. P2P computing, sometimes referred to as volunteer or global computing, utilizes idle CPU cycles of millions of personal computers for running various computations. Computational grids [1], on the other hand, are more structured ensembles of specialised resources such as scientific instruments, data repositories, clusters and supercomputers.

Computational grids are often made up of computing resources that are characterised by: (a) heterogeneity – where resources can vary in terms of architecture, operating system, and configuration; (b) dynamic availability – since there is no central administrative control over all participating resources, the availability of computing resources may change over time; and (c) use of private networks – clusters and NOWs are usually configured using NATs. The network that interconnects grid resources can vary from dedicated high-bandwidth backbones to the Internet. Large geographic distances mean that transient or permanent link failures are likely to take place. This dynamic nature of resources and its network infrastructure create unique challenges for executing grid applications.

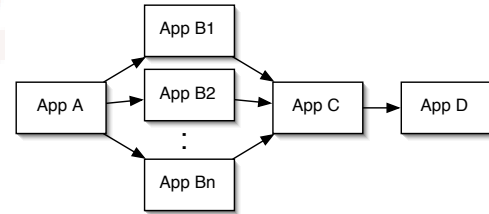


Fig. 1. A simple four-stage workflow. Arrows indicate data flow between component applications. Application B is an n-process parallel application.

Our research is motivated by the need to support communication in an emerging class of grid applications called scientific workflows. A workflow consists of multiple processing stages, where intermediate data generated in one stage are used by a subsequent stage for further processing. Typically, a stage in a workflow is an application that has been modified so that it is able to communicate with other stages. Thus, a grid workflow is a kind of computational/data processing pipeline, with data being captured, processed and manipulated throughout various stages (Fig. 1). Currently, the data transfers between component applications are realised in various ways: (a) using file transfers (e.g. GridFTP); (b) remote procedure call (e.g. RPC-V, GridRPC, OmniRPC, etc.); and (c) other custom-built mechanisms (e.g. Web Services).

We present the design and implementation of an asynchronous and persistent pipe mechanism, called π -channels. This mechanism is aimed at addressing the fault-tolerant issues with wide-area systems and has the following key features:

- 1) Simplified application coupling by employing a naming space called π -Space to identify communication endpoints with user-specified string names. This removes the need to specify the address of the receiver.
- 2) Decoupled communication between the source and the sink applications by persistence, which is achieved by adaptively caching π -channels and their data. Writers may proceed with pipe creation and generation of data, without regard for the status of the intended reader. This also enables a stored π -channel to be retrieved even when its writer has terminated.

- 3) Efficient communication is achieved by (a) using direct connections to stream data between source and sink; and (b) employing a communication thread that asynchronously handles incoming data.

The basic API for π -Space/ π -channels is presented along with the behavior of these operations. We discuss how fault-tolerance is provided by our mechanism under different conditions: (a) failure of the source or sink; and (b) link failure. The design of the server that implements the π -Space/ π -channels abstraction is presented including the underlying distributed algorithm. One key feature of our protocol is in its support for communication state transfers during migration. Essentially, it enables connections to be re-established with migrated components. Finally, we present some performance results from our benchmark application.

II. RELATED WORK

Interprocess communication mechanisms are fundamental building blocks for distributed applications. And over the years, many different communication models have been proposed. We briefly review the major models and differentiate our proposed mechanism.

A. Pipe/Channel Models

The pipe/channel model is probably the most well-known IPC mechanism. It is embodied in many mechanisms such as Unix pipes, named pipes, and TCP sockets [2]. Pipes are often used in Unix shell scripts, allowing easy composition of simple pipelines between processes. Socket programming with TCP, while used for implementing network applications, is too low-level for application-level 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 programming interface will generate a “broken pipe” exception, which has to be explicitly handled.

The channel abstraction has its early beginnings in Communicating Sequential Processes (CSP) [3]. CSP provides two basic channel operations: input (?) and output (!) and this simplicity has encouraged its adoption on current process calculi. Many coordination languages use the channel model, e.g. POLYLITH [4], Conic [5], etc. Recently, DP [6], [7] also employs a pipe-based communication mechanism for iterative grid applications. This system, however, has no support for persistence or fault-tolerance in the communication.

Communication persistence was previously explored by at least three papers: (a) persistent pipes for transactions commit [8]; (b) persistent connections (over TCP) that allow endpoint relocation without data loss [9]; and more recently, (c) the persistent streaming protocol that guarantees eventual delivery of data to receivers [10] and NapletSocket [11]. Unlike π -Spaces/ π -channels, no logical name space is provided by these for communication endpoint coupling. For example, IP addresses and ports are necessary for configuring endpoints in [8] and [9]. Furthermore, π -Spaces/ π -channels support asynchronous communication.

B. Message Passing

For general parallel programming on distributed-memory machines, message passing is the most popular approach. And the Message Passing Interface (MPI), with many implementations, is widely used for writing parallel programs. However, Grid applications like scientific workflows are typically composed of separate applications that has to be coupled. When coupling separate MPI applications, the MPI-2 standard provides Unix socket-like interface for accepting and establishing connections. When established between two groups of processes, an intercommunicator can then be used with `MPI_Send()` and `MPI_Recv()`.

The MPI communication model relies on a process rank as a means to identify processes. This is simple and elegant method and works very well on applications where the number of processes is known and fixed. This makes MPI well-suited for writing single-program-multiple-data (SPMD) applications. Newer models, such as that of Phoenix [12], modify the addressing scheme so that processes may join and leave the computation without the need to re-assign ranks.

C. Generative Communication Models

Linda tuplespace [13] is a classic example of a generative communication model that supports decoupled communication. In Linda, processes communicate by posting and retrieving ordered sequences of values called tuples onto a logical shared space called tuplespace. The elegance of this model of coordination served as inspiration to many current systems such as Sun’s Javaspace [14], and IBM’s T-spaces [15].

In practice, many applications benefit from the ability of an IPC mechanism to support efficient data transfers. On the traditional tuplespace models, such data transfers have to be performed through the shared space. This has lead to extending the model to support direct communication through some sort of streaming. One example is WCL [16], with its support for bulk transport of tuples. Taskspaces [17], [18] provide direct communication by using the space as a mechanism to identify communication endpoints (IP/port) and using TCP to connect processes. However, there is a lack of support for fault-tolerance during streaming.

III. π -CHANNELS: PERSISTENT PIPES

A π -channel is an enhanced unidirectional pipe that has a unique user-specified string name and a payload containing the data stream. Each pipe has one writer and 0 or more readers; and the payload is a FIFO sequence of arbitrary-length typed data segments, each treated as an indivisible unit. Fault-tolerant communication is achieved by persistence through adaptive caching. This persistence property enables π -channels to be created and written to at any time, independently of the sink/reader, thus encouraging temporal decoupling between readers and writers.

During π -channel creation, if the matching reader is known, a direct connection (if possible) is employed to efficiently transfer the pipe data. The π -channel effectively behaves like a TCP stream. Due to its persistence property, delivery of the

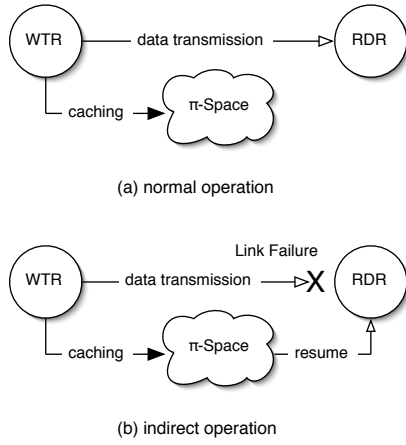


Fig. 2. π -channel behavior under two conditions: (a) when link is available and reader process (RDR) is able to receive; and (b) during a link failure, writer (WTR) continues to transmit data to the space, eventually when link is restored, delivery to the reader (RDR) resumes.

π -channel to the π -Space continues (Fig. 2a). This enables writing to the π -channel to proceed even if the link and/or reader has failed in the middle of communication. At this point, the π -channel data segments will continue to be directed towards the associated π -Space (Fig. 2b). When the link is eventually restored, delivery of the data can resume from the last point (either from the cache or from the writer).

A. π -channels: Programming Model and Semantics

The basic π -channels programming interface consists of the following primitives:

- 1) `pi_attach(str n, int s)`
attaches to a π -channel with the given name n on space identified by s . This primitive notifies the lookup server that this process has requested for the π -channel with name n . Even if the π -channel does not exist, this primitive does not block the caller. It returns a descriptor representing the “read” end of the π -channel. This operation also initiates asynchronous receive operation, allowing incoming segments to be received even before the `pi_read()` is issued.
- 2) `pi_create(str n, int s)`
creates a π -channel with the given name n on space s . On success, it returns a descriptor representing the “write” end of the created π -channel.
- 3) `pi_read(int d, ptr b, int len, dtype_t1 t)`
reads a data segment of len elements of type t into the memory block with starting address specified in b from the descriptor (d) that represents the “read” end of the attached π -channel. Because the `pi_attach()` is non-blocking, this primitive will block if there is no data currently available. It returns the number of elements successfully read or -1 if the end of the pipe.

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

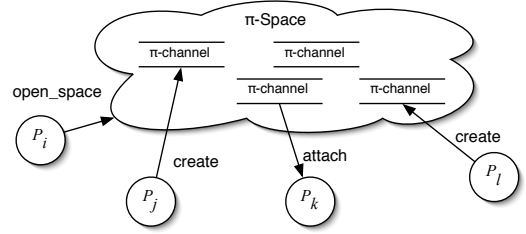


Fig. 3. Decoupled communication using π -channels between processes through a single π -Space.

- 4) `pi_write(int d, ptr b, int len, dtype_t t)`
writes a segment of len elements of type t from the memory block with starting address specified in b into the “write” end of the π -channel, represented by the descriptor d . It returns the number of elements written to the end of the π -channel.
- 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 .
- 7) `pi_seek(int d, int seg_id)`
moves the logical pipe pointer for d to the beginning of the segment identified by seg_id . This enables resuming reads/writes starting from some segment.
- 8) `pi_tell(int d)`
returns the current position (in segments) of the logical pipe pointer of the π -channel. The return value may be used in `pi_seek()` to resume reading/writing.
- 9) `pi_unlink(str n)`
marks the π -channel n as deleted, allowing the garbage collection. This has no effect for open π -channels.

B. π -Spaces: Abstraction of Shared Space for π -channels

Distributed processes communicate and coordinate with each other through one or more π -Spaces. Like `tuplespace`, a π -Space is a shared space abstraction for π -channels, which are explicitly posted and retrieved by processes (Fig. 3). A π -Space also presents a logical name space for π -channels, providing a dynamic lookup mechanism for runtime binding between π -channel endpoints.

The model of persistent π -channels fits elegantly with shared spaces. When a π -channel is created, a copy of it is automatically posted (or cached) on the specified π -Space. During a `pi_attach()`, the π -Space is accessed to retrieve the named π -channel. However, there is one caveat: within a single π -Space, a string name is bound to at most one π -channel. This disallows two or more distinct π -channel to have the same string name. This may appear restrictive, but it simplifies the semantics², e.g. during a `pi_attach()`, the

²And it also closely matches the semantics of conventional data files, if each π -channel name is treated as analogous to a full pathname.

name refers unambiguously to exactly one π -channel. To reuse an existing name, the existing π -channel has to be marked for deletion through the `pi_unlink()` primitive.

1) `int pi_mk_space(str sn, str pass)`

This primitive creates a π -Space with name `sn`, using a simple user-specified passphrase (`pass`) to control access to it. If multiple processes invoke this primitive, only one π -Space will be created.

2) `int pi_use_space(str sn, str pass)`

This primitive is invoked to gain access to a given π -Space `sn`. The passphrase `pass` has to match the one used to during the creation of this space. On success, this primitive returns a unique π -Space identifier.

3) `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.

C. Support for Fault-Tolerance and Application Migration

During a workflow execution, one or more of the following events may occur:

- 1) The source application (writer) leaves the workflow. Unless the dependent sink applications do not require any further data from this application, the entire workflow may be stalled until this application is restarted.
- 2) The sink application (reader) leaves the workflow. The writer continues streaming pipe data to the cache, ignoring the fact that the reader is no longer present.
- 3) The link between applications gets severed. Unless the π -Server itself crashes³, this is treated as the 2nd case.

We further distinguish two situations when an application leaves a workflow: (a) voluntary; and (b) involuntary. A voluntary departure can be treated as a migration from the current resource to another one, possibly decided by a resource manager⁴. We employ a communication state transfer (similar to [19]) mechanism to deal with voluntary migration of peers, re-establishing the connection when the migrated application rejoins the workflow. An involuntary departure may be due to either the machine failure, or the application has crashed. The scheduler will eventually detect this and restart the application on another available resource. We consider this as a failure of the application (under the fail-stop model, i.e., the application crashes and performs no further communication).

The third event is usually treated as a failure of the application. It can be detected when some communication operation cannot be completed, e.g. a broken pipe exception. In our implementation, each participating process uses a heartbeat algorithm to periodically poll all peers that it is communicating with. If a number of unacknowledged heartbeats is reached, the peer will consider as failed or unreachable. If this is

³The π -Server assumed to be 100% reliable and is run on an independent machine, similar to stable store concept in transactional databases.

⁴The discussion of the resource manager and scheduler is beyond the scope of this paper. Our mechanism is designed to run on existing schedulers like PBS and other resource managers.

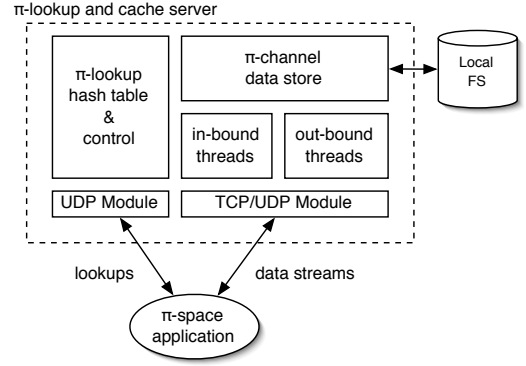


Fig. 4. Components of the π -Server

due to congestion and the peer eventually returns, a recovery mechanism is employed to reconnect this peer.

Each application is assumed to be piece-wise deterministic, so when it is restarted, it will perform the same sequence of π -channel operations. Most likely, many of the π -channels were already generated during the previous execution and are stored in π -Space. When the restarted application performs a `pi_create()` on an existing but incomplete π -channel, the primitive “simulates” all the `pi_write()` as if they are updating the cache, but no communication actually takes place. So the failure of any component application can be handled by simply restarting it. Moreover, our system does not exclude the use of checkpointing mechanisms. A custom-made checkpointing mechanism can use `pi_tell()` and `pi_seek()` to “fast-forward” the π -channels and resume normal operations.

IV. DESIGN AND IMPLEMENTATION

The π -channel and π -Space abstractions are implemented using an API with a corresponding client-side runtime system and a multi-threaded server process that provides the dynamic lookup and storage of π -channels. Each server process manages all π -channels that are created and accessed from within a single π -Space. In other words, there is a one-to-one mapping between server instances and π -Spaces. This encourages the user to employ multiple servers (and multiple π -Spaces) to improve the distribution of load among servers. For example, in large application that uses many π -channels, several π -Spaces(servers) may be configured. On a multi-cluster system, a π -Server may be deployed at the head nodes of each cluster, each serving as a communication gateway to processes on remote clusters.

A. Basic Components and General Functional Description

In Fig. 4, the components of the π -Server are presented. The server maintains a thread pool for managing incoming and outgoing transmissions. This server behaves like a non-archival file storage server, where π -channel data are buffered temporarily. Our implementation allows configuration for multiple threads to be employed, providing the necessary support to handle multiple readers and writers. The in-bound threads

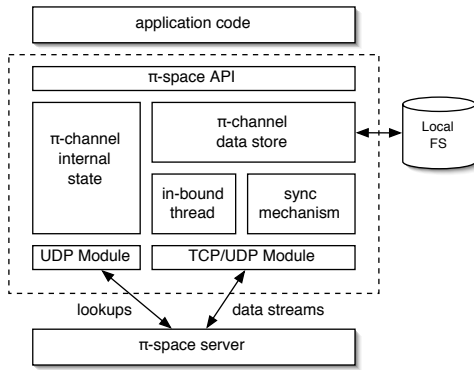


Fig. 5. Client-side Components

handle incoming data streams from writers (sources). The out-bound threads forward π -channel data to readers (sinks).

In Fig. 5, we present the components linked with the client application code. Each π -Space/ π -channel application has the basic components of the server, allowing it to accept incoming transmissions in an asynchronous manner. This is one key feature of our communication model. During a `pi_attach()`, the reader informs the server of its identity/address. When the server or the writer is generating the stream, it can forward the data to the reader even if the latter is busy performing other tasks, e.g. computations, I/O, etc. All incoming transmissions are handled by a single in-bound thread⁵ and data received for each π -channel are stored in a buffer (through the data store component). If the buffer capacity is exceeded, data segments are automatically stored into a local file, identified by the `pipe_id`. When the client code performs the `pi_read()`, this buffer is accessed to retrieve the requested data segment.

B. General Description of the Distributed Algorithm

In Fig. 6, we present the distributed algorithm for implementing the API and the server. For conciseness, we adopt the Python convention of using indentation to indicate block structure (i.e., statements that are aligned belong to the same block). Throughout this section, when specific statements in our algorithm is referenced, we will use the notation (**#line_no**).

Every π -channel is uniquely identified by the integer pair: (`pipe_id`, `space_id`). We remove the reference to the `space_id` since the algorithm is in the context of a single π -Space. Within a single π -Space, the `pipe_id` is unique and is associated with the unique name of the π -channels.

Each participating application has the following variables:

- 1) `my_id` – unique ID / address of each participating application. In our current TCP/IP implementation, this is an IP/port pair. This port number is used for incoming connections from the π -Server or peers and is dynamically generated during application startup.
- 2) `id.tab` – a local table (**#4**, **#14**) for associating the `pipe_id` with open file descriptors (`fd.list`), `pipe`

⁵This does not mean that only one active connection is accepted. The thread is event-driven and allows up to 1024 simultaneously active connections.

read-write pointers, and other state information. The position of the entry on this table is used as the descriptor of a π -channel within a single process/application. This descriptor is not to be confused with the system-wide unique `pipe_id`. This table corresponds to the π -channel internal state component in Fig. 5.

During a `pi_create()`, a `put` request (**#10**) is sent to the π -Server managing the π -Space on which this π -channel is to be created. It (**#58–#65**) creates an entry for this π -channel on a hash table and returns a space-wide unique pipe ID (`pipe_id`), as reference. It (**#58**, **#61**, **#62**) also replies with a list (possibly empty) of destinations to the requesting process. This is to identify the destination of the payload of the π -channel. By default, if the reader's identity is known, the reader's address is included as the first entry in this list. The server's address is appended into this list.

The `pi_create()` establishes (**#12**, **#13**) a TCP connection with the list of destinations and keeps the file descriptors associated with the `pipe_id`. It returns as descriptor (**#14**, **#16**) the position of the entry in `id.tab` for this newly-created π -channel. This descriptor is used by `pi_write()` to reference (**#27–#35**) the table and obtain the TCP file descriptors to write the data segments to. This write will at least involve sending the data to a server that corresponds to the π -Space.

The `pi_attach()` sends a `get` request to the server (**#2**), which replies (**#51**, **#52**) with the unique `pipe_id` for the requested π -channel, even if it does not exist yet. The server creates an entry for this π -channel, storing the address of the client. This address is used later when the corresponding writer issues a `pi_create()`. During the `put` request, the address of the reader will be sent to the writer (**#61**), enabling the writer to connect to the reader.

In `pi_read()`, there is no operation that directly accesses the open connection with the writer. This is because incoming data segments are handled by the in-bound thread (**#66–#74**). This thread performs the TCP listen and accept operations on behalf of the application. The π -Server also maintains a pool of these threads for the same purpose.

When the in-bound thread accepts an incoming connection, it creates a buffer (**#70**) to store the inbound data segments for this π -channel. Each incoming data is received as segments (**#72–#74**), containing type information (not shown), length, type information, offset, and the pipe reference ID (`pipe_id`). The received segments are stored in a buffer, which is shared with the client's main thread, so that `pi_read()` can retrieve (**#19**, **#22**) the data segment. This design allows an application to accept any incoming streams at any time during its execution, even when it is performing computations. Thus, the ability to support asynchronous communication.

The out-bound thread forwards π -channel data to readers (**#54**, **#75–#78**). This takes place when a `get` request is received by the server and the π -channel data is already cached (even partially). When a sink process has crashed and resumes, this thread is used by the π -Server to send segments that the reader did not receive (in conjunction with the `pi_seek()` primitive).

Heartbeat messages (not shown) are used to determine if

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 in 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 (put,&pipe_id,&dest_list) from  $\pi$ -Server
12   foreach dest in dest_list do:
13     fd_list[i]  $\leftarrow$  connect to dest
14   id  $\leftarrow$  free entry in 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 data[off, len]  $\in$  buffer || eoc
21   if data found:
22     get segment of len from buffer
23     store segment into m
24     return len
25   return -1 /* end of channel */

26 pi_write(id, buf, len):
27   pipe_id  $\leftarrow$  map pipe id from id
28   fd_list  $\leftarrow$  retrieve fdlist for id
29   off  $\leftarrow$  compute offset
30   success_count  $\leftarrow$  0
31   foreach dest in fd_list do:
32     write SEG (pipe_id, buf, 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 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 in  $\pi$ -table
47   if entry does not exist:
48     entry  $\leftarrow$  create n in  $\pi$ -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.cached:
54     initiate forwarding to src

55 On (put,&n,&src):
56   entry  $\leftarrow$  find n in  $\pi$ -table
57   if entry does not exist:
58     entry  $\leftarrow$  create n in  $\pi$ -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.cached  $\leftarrow$  True
65   send (put,pipe_id,dest_list) to src

```

Algorithm of the in-bound thread

```

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

72 On data segment SEG (&pipe_id,&buf,&offset,&len):
73   buffer  $\leftarrow$  find local buffer for pipe_id
74   put segment buf at offset offset

```

Algorithm of the out-bound thread

```

75 if forwarding a pipe pipe_id to dest:
76   fd  $\leftarrow$  open a connection to dest
77   foreach segment in the local buffer:
78     send segment to fd

```

Fig. 6. Distributed Algorithm for π -Space/ π -channels.

peers are still active. Incorporating this is straightforward since our implementation is multi-threaded. When an in-bound thread receives this heartbeat, it replies with an *alive* message, indicating its status. Heartbeats also determine if a source or a sink will be migrated. Due to lack of space, we only outline the key steps of the migration mechanism (Fig. 7). For example, it is assumed that some checkpoint of the application state is saved and restored after migration. The basic idea of the code in (#79–#86) is to attempt a graceful shutdown of all open connections before migrating. Since pipes are cached, undelivered data segments can be later retrieved from the π -Server. The *hold_list* (#89, #44) contains the addresses of migrating process, to ensure that no further communication with these processes take place.

The communication state migration (similar but simpler than that of [19]) performs a connection hand-over by the writer

with the newly-migrated instance of the reader (#39–#43). In Fig. 8(b), we show that with migration of communication state, the effect is that the newly-migrated peer can re-establish connection with the writer so that: (1) Seg 2 is retrieved from the π -Server; and at the same time (2) Seg 3 is directly streamed from the active writer. This reduces the time for the reader to receive the entire data stream.

C. Incorporating Replication for Resilience and Other Issues

Replication may be used to improve the fault tolerance of the algorithm. A troupe of servers may be employed to manage a single π -Space by replicating the state information for the π -channels. If there is a need for multiple π -Space, then multiple troupes are used. The main modification will be in the *pi_attach()* and *pi_create()* primitives. In this case, these functions will send the *get* and *put* requests to all

Outline of Communication State Migration

```

79  If I am migrating:
80    disable sending acks for heartbeats
81    migrating ← True
82    foreach open  $\pi$ -channel:
83      save offset for  $\pi$ -channel to 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

```

Fig. 7. Outline of the Communication Migration Protocol.

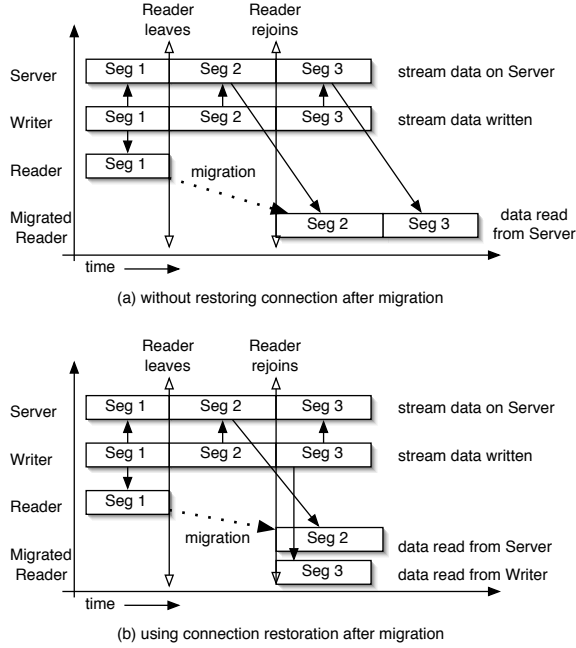


Fig. 8. Concurrently reading of a π -channel from π -Server and writer. In (a), upon migration of the reader to a new location, it resumes reading the rest of the unread pipe from the cache. In (b), upon its resumption to read from the server, it also resumes connection from the writer. This allows for the concurrent read (using in-bound thread) from two sources: the cache and the writer. Solid arrows indicate flow of data, while the dotted arrow represents migration of the reader. The x-axis is time and the y-axis is space.

servers. A quorum-based approach may be employed so that put requests are performed on all replicas while get requests can be served by any single replica.

During TCP connection establishment, a check is first made to determine if an active connection with that peer exists. If it exists, this connection is reused. If this connection is not longer active, a new connection will be attempted. When a π -channel is closed for writing or reading, the TCP connection between the communicating peers is not reset.

V. EXPERIMENTAL RESULTS

We conducted various experiments to evaluate our implementation. In Table I, we present some relevant details of the facilities we used for conducting the experiments. VPAC

Name	Location	Type	Processor	#CPUs
nexus.csse...	Monash	Server	Sparc	1
mahar.csse...	Monash	Cluster	Intel P4	50
brecca.vpac...	VPAC	Cluster	Intel P4	194
wexstan.vpac...	VPAC	Cluster	AMD Opteron	246

TABLE I

PARTICIPATING SYSTEMS IN THE EXPERIMENTS.

Number of Clients	1	2	4	8
Execution Times	6.7s	7.5s	11.5s	19.5s
Request Rate	3731	3333	2174	1282

TABLE II

UDP REQUEST-REPLY PERFORMANCE ON LAN SETUP.

stands for the Victorian Partnership for Advanced Computing, a consortium of universities in the state of Victoria, Australia.

Considering that the π -Server will be expected to handle multiple clients, we evaluated how it is able to deal with flooding of requests. In Table II, we present the performance of the UDP component of our implementation. The UDP mechanism implements a lightweight protocol for retransmission of lost messages. The execution times are obtained from the average times of the clients tested. Each client conducting 25000 lookups. The clients are executed on *brecca* while the server ran on *wexstan*. Each lookup used a randomly-selected π -channel name from a pool of 100.

In Table III, we present the timings for performing UDP lookups on a WAN environment (between Monash and VPAC). Like in the LAN setup, clients are run on *nexus* while the server on *wexstan*. The times are the average execution time of clients, each performing 5000 lookup requests. These results indicate that the bottleneck for grid applications will most likely be the high latencies between the π -Server and the clients. Secondly, it is unlikely that in a real application, the requests to lookup will take place in rapid succession. This is because a lookup is only necessary to match the reader with the writer of a π -channel, i.e., at *pi.attach()* and *pi.create()*. It is expected that most of the time will be spent in sending and retrieving the pipe data segments.

In Fig. 9, we present the rate of sending message segments between a pair of applications. The π -Server is run on *wexstan*, one application on *brecca* and the other on *nexus*. The corresponding throughput results are presented in Fig. 10. These results show that the caching overhead is minimal, although this will likely increase if multiple replicated

Number of Clients	1	2	4	8
Execution Times	7.55s	7.85s	8.5s	12.5s
Request Rate	662	636	588	400

TABLE III

UDP REQUEST-REPLY PERFORMANCE ON WAN SETUP.

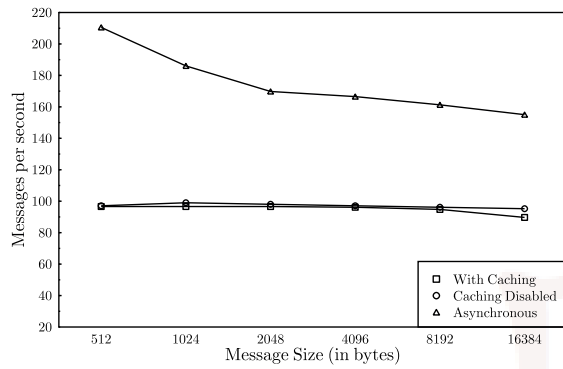


Fig. 9. Measured message-send rates with our ping-pong benchmark.

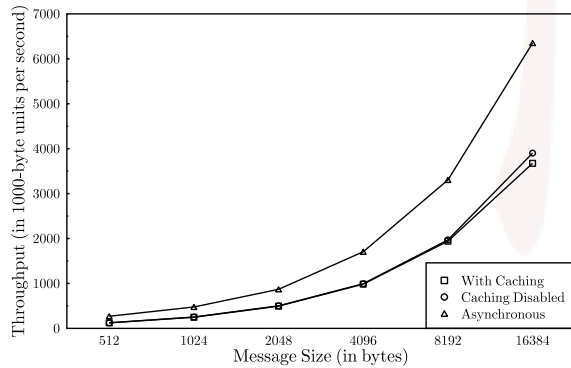


Fig. 10. Measured throughput with our ping-pong benchmark.

servers are employed. The best performance is achieved when asynchronous operation is used. This is made possible when the in-bound threads are enabled to receive message segments while the application is sending segments to the peer. The overlap in sending and receiving results in a better utilisation of the available bandwidth.

VI. CONCLUSIONS

We have presented the design and prototype implementation of π -Space/ π -channels, a communication mechanism suitable for scientific workflows on dynamic environments like grids, where resources may fail and network links may be disrupted. 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 our caching mechanism incurs minimal overhead and when asynchronous operation is employed, throughput can be substantially increased. With asynchronous operation, sending and receiving of messages can be overlapped, yielding up to double the message sending rates than normal non-asynchronous operation.

ACKNOWLEDGMENT

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] I. Foster and C. Kesselman, "Computational Grids," in *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 15–51.
- [2] W. R. Stevens, *Unix Network Programming: Networking APIs: Sockets and XTI*, 2nd ed. Prentice-Hall PTR, 1998, vol. 1.
- [3] C. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.
- [4] J. M. Purtilo, "The POLYLITH Software Bus," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 1, pp. 151–174, January 1994.
- [5] J. Magee, J. Kramer, and M. Sloman, "Constructing Distributed Systems in Conic," *IEEE Transactions in Software Engineering*, vol. 15, no. 6, pp. 663–675, June 1989.
- [6] 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*, vol. 12, no. 10, pp. 1052–1065, October 2001.
- [7] 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.
- [8] 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*. Washington, DC, USA: IEEE Computer Society, 1991, pp. 286–293.
- [9] 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*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 300–307.
- [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*, vol. 1. Washington, DC, USA: IEEE Computer Society, 2004, pp. 529–533.
- [11] X. Zhong and C.-Z. Xu, "A Reliable Connection Migration Mechanism for Synchronous Transient Communication in Mobile Codes," in *Proc. of the 2004 International Conference on Parallel Processing*. IEEE Press, 2004.
- [12] K. Taura, K. Kaneda, T. Endo, and A. Yonezawa, "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*. New York, NY, USA: ACM Press, 2003, pp. 216–229.
- [13] N. Carriero and D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," *ACM Computing Surveys*, vol. 21, no. 3, pp. 323–357, September 1989.
- [14] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpacesTM Principles, Patterns, and Practice*. Addison-Wesley, 1999.
- [15] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford, "T Spaces," *IBM Systems Journal*, vol. 37, no. 3, pp. 454–474, 1998.
- [16] A. Rowstron, "WCL: A Co-ordination Language for Geographically Distributed Agents," *World Wide Web*, vol. 1, no. 3, pp. 167–179, September 1998.
- [17] 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*. New York, NY, USA: ACM Press, 2003, pp. 1024–1030.
- [18] 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.
- [19] K. Chanchio and X.-H. Sun, "Communication State Transfer for Mobility of Concurrent Heterogeneous Computing," *IEEE Transactions on Computers*, vol. 53, no. 10, pp. 1260–1273, October 2004.