coordinated manner so that it can be restarted at the checkpoint on a different set of cluster nodes at a later time. In checkpointing and restarting a distributed application, ZapC separates the processing of network state from per node application state. ZapC only requires synchronized operation in checkpointing the network state, which represents a small fraction of the overall checkpoint time. Checkpoint-restart operations on per node application state are designed to proceed in parallel with minimal synchronization requirements among nodes, resulting in faster checkpoint and restart times. ZapC can also directly stream checkpoint data from one set of nodes to another, enabling direct migration of a distributed application to a new set of nodes without saving and restoring state from secondary storage.

ZapC uniquely supports complete checkpoint-restart of network state in a transport protocol independent manner without application or library support. It leverages the socket abstraction and correctly saves and restores all socket state, including socket parameters, socket data queues, and minimal protocol specific state. ZapC accomplishes this in a portable manner using the standard socket interface without detailed knowledge of the underlying network protocol data structures. ZapC accounts for network state in a protocol independent manner for reliable and unreliable network protocols, including TCP, UDP and raw IP.

We have implemented a ZapC prototype that runs across multiple Linux operating system versions, including both Linux 2.4 and 2.6 kernels, and demonstrated its effectiveness on cluster systems in checkpointing and restarting a number of real distributed application workloads, including both MPI and PVM applications. Our results show that ZapC imposes very low virtualization overhead and provides fast checkpoint-restart times for unmodified distributed network applications. We show that ZapC uniquely provides distributed checkpoint-restart functionality without any application, library, kernel, or network protocol modifications.

This paper focuses on the design and implementation of the ZapC checkpoint-restart mechanism for distributed applications. Section 2 describes related work. Section 3 provides an overview of the ZapC system architecture. Section 4 presents the ZapC distributed checkpoint-restart scheme. Section 5 discusses ZapC support for checkpoint-restart of network state. Section 6 presents performance results using ZapC on a number of distributed network applications. Finally, we present some concluding remarks.

2 Related Work

Many application checkpoint-restart mechanisms have been proposed [25, 28]. Application-level checkpoint-restart mechanisms are directly incorporated into the applications, often with the help of languages, libraries, and preprocessors [11, 20]. These approaches are generally the most efficient, but they are not transparent, place a major burden on the application developer, may require the use of nonstandard programming languages, and cannot be used for unmodified or binary-only applications.

Library checkpoint-restart mechanisms [26, 36] reduce the burden on the application developer by only requiring that applications be compiled or relinked against special libraries. However, such approaches do not capture important parts of the system state, such as interprocess communication, network sockets and threading.

Library-level distributed checkpoint-restart mechanisms resort to substituting standard message-passing middleware, such as MPI [5] and PVM [6], with specialized checkpoint-aware middleware versions. [29] provides a good survey. To perform a checkpoint they flush all the data communication channels to prevent loss of in-flight messages. Upon restart they reconstruct the network connectivity among the processes, and remap location information according to how the network addresses have changed. Examples include MPVM (MIST) [13], CoCheck [27], LAM-MPI [30], FT-MPI [17], C^3 [12], Starfish [7], PM2 [35] and CLIP [15]. These library-level approaches require that applications be well-behaved. They cannot use common operating system services as system identifiers such as process identifiers cannot be preserved after a restart. As a result, these approaches can only be used for a narrow range of applications.

Operating system checkpoint-restart mechanisms utilize kernel-level support to provide greater application transparency. They do not require modification to the application source code nor relinking of the application object code, and typically allow a checkpoint at any time. Earlier approaches [22, 23, 31] required specialized operating systems or invasive kernel modifications. More recent approaches such as CRAK [37] and its successor Zap [24] have been designed as loadable kernel modules that can be used with unmodified commodity operating systems. These systems provide checkpoint-restart only for applications running on a single node. ZapC builds on our previous work on Zap to provide general coordinated checkpoint-restart of distributed network applications running on commodity multiprocessor clusters.

Unlike other operating system checkpoint-restart mechanisms, BLCR [16] and Cruz [21] provide support for coordinated checkpoint-restart of distributed applications. However, BLCR does not support checkpoint-restart of socket state and relies on modifying applications to cooperate with checkpoint-aware message passing middleware [16]. BLCR also cannot restart successfully if a resource identifier is required for the restart, such as a process identifier, is already in use.

Cruz [21] also builds on Zap, but is based on an out-

dated and incomplete Linux 2.4 implementation. It restricts applications to being moved within the same subnet and requires unique, constant network addresses for each application endpoint. It cannot support different network transport protocols but instead uses low-level details of the Linux TCP implementation to attempt to save and restore network state to provide checkpoint-restart of distributed network applications in a transparent manner. This is done in part by peeking at the data in the receive queue. This technique is incomplete and will fail to capture all of the data in the network queues with TCP, including crucial out-of-band, urgent, and backlog queue data. No quantitative results have been reported to show that Cruz can restart distributed applications correctly.

3 Architecture Overview and Pods

ZapC is designed to checkpoint-restart an entire distributed network application running on a set of cluster nodes. It can be thought of in terms of three logical components: a standalone checkpoint-restart mechanism based on Zap that saves and restores non-network per-node application state, a manager that coordinates a set of agents each using the standalone checkpoint-restart mechanism to save and restore a distributed application across a set of cluster nodes in a consistent manner, and a network checkpoint-restart mechanism that saves and restores all the necessary network state to enable the application processes running on different nodes to communicate. For simplicity, we describe these ZapC components assuming a commodity cluster in which the cluster nodes are running independent commodity operating system instances and the nodes all have access to a shared storage infrastructure. For example, a common configuration would be a set of blade servers or rackmounted 1U servers running standard Linux and connected to a common SAN or a NAS storage infrastructure.

ZapC's standalone checkpoint-restart mechanism uses the pod (PrOcess Domain) virtual machine abstraction previously introduced in Zap. To execute a distributed application across a set of cluster nodes, ZapC encapsulates the application processes running on each node in a pod to decouple those processes from the underlying host. Unlike a traditional operating system, each pod provides a self-contained unit that can be isolated from the system, checkpointed to secondary storage, migrated to another machine, and transparently restarted. This is made possible because each pod has its own virtual private namespace, which provides the only means for processes to access the underlying operating system. To guarantee correct operation of unmodified applications, the pod namespace provides a traditional environment with unchanged application interfaces and access to operating system services and resources.

Operating system resource identifiers, such as process IDs (PIDs), must remain constant throughout the life of a process to ensure its correct operation. However, when a process is moved from one operating system instance to another, there is no guarantee that the destination system will provide the same identifiers to the migrated process; those identifiers may in fact be in use by other processes in the system. The pod namespace addresses these issues by providing consistent, virtual resource names. Names within a pod are trivially assigned in a unique manner in the same way that traditional operating systems assign names, but such names are localized to the pod. Since the namespace is virtual, there is no need for it to change when the pod is migrated, ensuring that identifiers remain constant throughout the life of the process, as required by applications that use such identifiers. Since the namespace is private to a given pod, processes within the pod can be migrated as a group, while avoiding resource naming conflicts among processes in different pods. ZapC transparently remaps pod virtual resources to real operating system resources as a pod migrate from one node to another. For example, ZapC only allows applications in pods to see virtual network addresses which are transparently remapped to underlying real network addresses as a pod migrates among different machines. This enables ZapC to migrate distributed applications to any cluster regardless of its IP subnet or addresses.

Pod namespaces are supported using a thin virtualization layer based on system call interposition mechanism and the `chroot` utility with file system stacking to provide each pod with its own file system namespace. ZapC's pod virtualization layer is designed to be implemented entirely in a dynamically loadable kernel module. A key issue that is addressed is the lack of atomicity of system call interposition when implemented in a kernel module separate from the base kernel. To provide multiprocessor support for virtualization, ZapC provides a set of low overhead reference counts to address potential race conditions that can occur in multiprocessor systems.

ZapC combines pod virtualization with a pod checkpoint-restart mechanism that employs higher-level semantic information specified in an intermediate format rather than kernel specific data in native format to keep the format portable across different kernels. ZapC by default assumes a shared storage infrastructure across cluster nodes and does not generally save and restore file system state as part of the pod checkpoint image to reduce checkpoint image size. Instead, ZapC can be used with already available file system snapshot functionality [2, 19] to also provide a checkpointed file system image. Further details on the pod virtualization and checkpoint-restart mechanisms are discussed elsewhere [10, 24, 34].

With ZapC, a distributed application is executed in a manner that is analogous to a regular cluster, ideally placing each application endpoint in a separate pod. For exam-

ple, on multiprocessor nodes that run multiple application endpoints, each endpoint can be encapsulated in a separate pod. To leverage mobility, it is advantageous to divide the application into many independent pods, since the pod is the minimal unit of migration. This allows for maximum flexibility when migrating the application. ZapC can migrate a distributed application running on N cluster nodes to run on M cluster nodes, where generally $N \neq M$. For instance, a dual-CPU node may host two application endpoints encapsulated in two separate pods. Each pod can thereafter be relocated to a distinct node; they do not need to be migrated together to the same node.

4 Distributed Checkpoint/Restart

To checkpoint-restart a distributed network application, ZapC provides a coordinated checkpoint-restart algorithm that uses the pod checkpoint-restart mechanism and a novel network state checkpoint-restart mechanism described in Section 5. We assume that all the network connections are internal among the participating nodes that compose the distributed application; connections going outside of the cluster are beyond the scope of this paper. Although ZapC allows multiple pods to execute concurrently on the same node, for simplicity, we describe ZapC operation below assuming one pod per node.

Our coordinated checkpointing scheme consists of a *Manager* client that orchestrates the operation and a set of *Agents*, one on each node. The Manager is the front-end client invoked by the user and can be run from anywhere, inside or outside the cluster. It accepts a user's checkpoint or restart request and translates it into a set of commands to the Agents. The Agents receive these commands and carry them out on their local nodes.

The Manager maintains reliable network connections with the Agents throughout the entire operation. Therefore an Agent failure will be readily detected by the Manager as soon as the connection becomes broken. Similarly a failure of the Manager itself will be noted by the Agents. In both cases, the operation will be gracefully aborted, and the application will resume its execution.

A checkpoint is initiated by invoking the Manager with a list of tuples of the form $\ll$`node`, `pod`, `URI`$\gg$. This list specifies the nodes and the pods that compose the distributed application, as well as the destination for the checkpointed data (`URI`). The destination can be either a file name or a network address of a receiving Agent. This facilitates direct migration of an application from one set of nodes to another without requiring that the checkpoint data first be written to some intermediary storage.

The Manager and the Agents execute the checkpoint algorithms given in Figure 1. Given a checkpoint request, the Manager begins with broadcasting a `checkpoint` command to all participating nodes. Upon receiving the command, each Agent initiates the local checkpoint procedure, that is divided into four steps: suspending the designated pod, invoking the network-state checkpoint, proceeding with the standalone pod checkpoint, and finalizing the checkpoint. The Agent also performs three companion steps, `2a`, `3a`, and `4a` in Figure 1, which are not directly related to the local checkpoint procedure, but rather to its interaction with the Manager. `3a` and `4a` both test the same condition, ensuring that the Agent only finishes after having satisfied two conditions: it has reported "done", and it received the *continue* message from the Manager.

Each Agent first suspends its respective pod by sending a `SIGSTOP` signal to all the processes in the pod to prevent those processes from being altered during checkpoint. To prevent the network state from changing, the Agent disables all network activity to and from the pod. This is done by leveraging a standard network filtering service to block the links listed in the table; Netfilter [3] comes standard with Linux and provides this functionality. The Agent then obtains the network *meta-data* of the node, a table of $\ll$`state`, `source`, `target`$\gg$ tuples showing all network connections of the pod. This is the first information saved by the Agent as part of the checkpoint and is used by the restart procedure to correctly reconstruct the network state. The `source` and `target` fields describe the connection endpoint IP addresses and port numbers. The `state` field reflects the state of the connection, which may be full-duplex, half-duplex, closed (in which case there may still be unread data), or connecting. The first three states are for established connections while the last state is a transient state for a not yet fully established connection.

```
Checkpoint Manager:
 1 Send 'checkpoint' to Agents
 2 Receive meta-data from Agents
 3 Send 'continue' to Agents
 4 Receive status from Agents

Checkpoint Agent:
 1  Suspend pod, block network
 2  Take network checkpoint
 2a Report meta-data to Manager
 3  Take standalone checkpoint
 3a If received 'continue' then
    unblock network
 4  Report 'done'
 4a while not received 'continue'
    wait for Manager
```

Figure 1. Coordinated checkpoint algorithms

Once the pod's network is frozen, the Agent checkpoints the network state. When finished, the Agent notifies the Manager that it has concluded its network state checkpoint, and reports its *meta-data*. It then proceeds to perform the

standalone pod checkpoint. The Agent cannot complete the standalone pod checkpoint until the Manager has received the *meta-data* from all participating Agents, at which point the Manager tells the Agents they can continue. ZapC checkpoints the network state before the other pod state to enable more concurrent checkpoint operation by overlapping the standalone pod checkpoint time with the time it takes for the Manager to receive the *meta-data* from all participating Agents and indicate that they can continue.

In the last step, the action taken by the Agent depends on the context of the checkpoint. If the application should continue to run on the same node after the checkpoint (i.e. taking a snapshot), the pod is allowed to resume execution by sending a `SIGCONT` to all the processes. However, should the application processes migrate to another location, the Agent will destroy the pod locally and create a new one at the destination site. In both cases, a file-system snapshot (if desired) may be taken immediately prior to reactivating the pod.

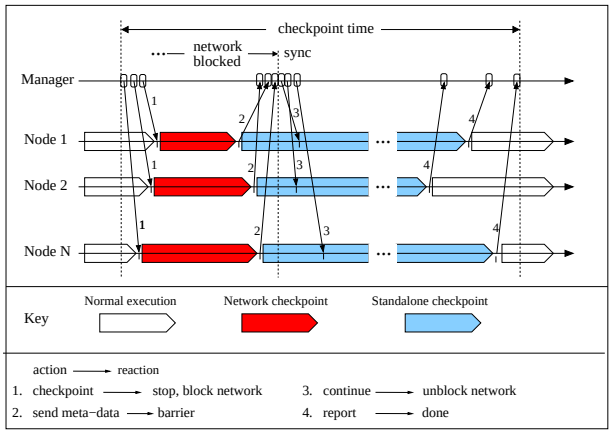


Figure 2. Coordinated checkpoint timeline

To provide a better understanding of the checkpoint timing requirements, Figure 2 illustrates a typical checkpoint timeline. The timeline is labeled with numbers that correspond to the steps of the checkpoint algorithm as described in Figure 1. The timeline shows that the that the entire checkpoint procedure executes concurrently in an asynchronous manner on all participating node for nearly its entire duration. Figure 2 shows the only synchronization point is the "sync" at the Manager after step 2 and during step 3.

This single synchronization is necessary and sufficient for the checkpoint procedure to be coherent and correct. It is necessary for the Agents to synchronize at the Manager before completing their standalone pod checkpoints and unblocking their networks. Otherwise it would be possible for one node to resume operation, re-engage in network activity, and deliver data to another node that had not begun its checkpoint. This would result in an inconsistent global state, as the state of the latter node will contain data that is not marked as sent in the already-saved state of the former.

The single synchronization is sufficient since every pod ensures consistency by blocking its connections independently of other pods. Once a pod has blocked its connections, there is no interaction with any other pod even if the network of other pods is not yet blocked. The pod is already isolated and does not need to wait for all other pods to block their connections. By not having to wait for other pods initially, the network activity is only blocked for the minimal required time.

A restart is initiated by invoking the Manager with a list of tuples of the form $\ll$`node`, `pod`, `URI`$\gg$. This list describes the mapping of the application to nodes and pods, where `URI` indicates the location of the checkpoint data. A key requirement of the restart is to restore the network connections of the distributed application. A naive approach would be to manually create the internal kernel data structures and crowd them with the relevant data, but this is not easy and requires intimate knowledge of the protocol implementation, tight cooperation between the peers, and careful adjustments of protocol-dependent parameters. Since ZapC is restarting the entire distributed application, it controls both ends of each network connection. This makes it straightforward to reconstruct the communicating sockets on both sides of each connection using a pair of `connect` and `accept` system calls. This leverages the standard socket interface for creating network connections and results in a robust, easy to implement and highly portable approach.

Using this approach, the Manager and the Agents execute the restart algorithms given in Figure 3, which are similar to the checkpoint counterparts. Given a restart request, the Manager begins sending a `restart` command to all the Agents accompanied by a modified version of the *meta-data*. The *meta-data* is used to derive a new network connectivity map by substituting the destination network addresses in place of the original addresses. This will outline the desired mapping of the application to nodes/pods pairs. In the case of a restart on the same set of nodes (e.g. recovering from a crash), the mapping is likely to remain unmodified. In the case of migration, the mapping will reflect the settings of the alternate execution environment, particularly the network addresses at the target cluster.

As part of the modified *meta-data*, the Manager provides a schedule that indicates for each connection which peer will initiate and which peer will accept. This is done by tagging each entry as either a *connect* or *accept* type. This is normally determined arbitrarily, except when multiple connections share the same source port number. Source port numbers can be set by the application if not already taken or assigned automatically by the kernel; specifically when a TCP connection is accepted, it inherits the source port

```
Restart Manager:
 1 Send 'restart' and
   meta-data to Agents
 2 Receive status from Agents

Restart Agent:
 1  Create a new pod
 2  Restore network connections
 3  Invoke network restart
 4  Invoke standalone restart
 5  Report 'done'
```

Figure 3. Coordinated restart algorithms

number from the "listening" socket. To correctly preserve the source port number when shared by multiple connections, these connections must be created in a manner that resembles their original creation, as determined by the above schedule.

The Agents respond to the Manager's commands by creating an empty pod into which the application will be restored. It then engages the local restart procedure, which consists of three steps: recovering the network connectivity, restoring the network state, and executing the application standalone restart. Once completed, the pod will be allowed to resume execution without further delay.

The recovery of the network connectivity is performed in user space and is fairly straight forward. The *meta-data* that the Agent received from the Manager completely describes the connectivity of the pod, and can be effectively used as a set of instructions to re-establish the desired connections. The Agent simply loops over all the entries (each of type *connect* or *accept*), and performs the suitable action. If the *state* field is other than full-duplex, the status of the connection is adjusted accordingly. For example, a closed connection would have the `shutdown` system call executed after the rest of its state has been recovered.

Generally, these connections cannot be executed in any arbitrary order, or a deadlock may occur. Consider for instance an application connected in a ring topology (each node has two connections - one at each side): a deadlock occurs if every node first attempts to accept a connection from the next node. To prevent such deadlocks, rather than using sophisticated methods to create a deadlock-free schedule, we simply divide the work between two threads of execution. One thread handles requests for incoming connections, and the other establishes connections to remote pods. Hence, there is no specific order at which connections requests should arrive at the Agent. The result is a simple and efficient connectivity recovery scheme, which is trivial to implement in a portable way.

Once the network connectivity has been re-established, the Agent initiates the restart of the network-state. This ensures that we reinstate the exact previous state of all network connections, namely connection status, receive queue, send queue and protocol specific state. Similarly to the distributed checkpoint, the motivation for this order of actions is to avoid forced synchronization points between the nodes at later stages. In turn, this prevents unnecessary idle time, and increases concurrency by hiding associated latencies. With this framework, the only synchronization that is required is indirect and is induced by the creation of network connections. As demonstrated in Section 6, the standalone restore time greatly dominates the total restore time, and fluctuates considerably. Positioning it as the first to execute may lead to imbalances and wasted idle time due to the synchronization that follows. Instead, our scheme manages to both minimize the loss by doing it early, and enable the pods continue their execution as soon as they conclude their standalone restart.

A key observation about our restart scheme is that it does not require that the network be disabled for any intermediate period. Recall that with checkpoint, the network was shut off to ensure a consistent state. The challenge was to capture the state of live connections that already carry data in the queues, and are likely to be transient. Conversely the re-established network connections are entirely controlled by out restart code. It is guaranteed that no data, but that which we choose to explicitly send, will be transmitted through the connection, until the application resumes execution (which will only occur at the end of the restart).

The final notch of the procedure is the standalone restart, invoked locally by each Agent after the network state has been successfully restored. To conclude the entire operation, each Agent sends a summary message to the Manager, specifying the completion status (failure or success) and the name of the new pod that has been created. The Manager collects this data from all the Agents and reports it back to the user.

5 Network-State Checkpoint/Restart

The *network-state* of an application is defined by the collection of the network-states of its communication endpoints. From the application's standing point, the primary abstraction of a communication endpoint is a *socket*. A socket is associated with a network protocol upon creation. The application can bind a socket to an address, connect to an address, accept a connection, as well as exchange data. The operating system in turn, keeps a certain amount of state for each socket. The network-state checkpoint-restart is responsible for capturing and restoring this state.

The state of a socket has three components: socket parameters, socket data queues and protocol specific state. The socket parameters describe socket properties related to its state, e.g. connected or not, and to its behavior, e.g. blocking or non-blocking I/O. The data queues - specifi-

cally send and receive queues, hold incoming and outgoing data respectively, that is handle by the network layer. Protocol specific data describes internal state held by the protocol itself. For instance, TCP connection state and TCP timers are part of its state.

Saving the state of the socket parameters is fairly straightforward. Recall that while taking a network-state checkpoint, the processes in the pod are suspended and cannot alter the socket state. Also, the network is blocked and is only restarted later on after all the applications involved in the checkpoint have terminated their local network-state checkpoint. That given, the socket parameters can be safely extracted at this point. Furthermore, these properties are user-accessible via a standard interface provided by the operating system, namely `getsockopt` and `setsockopt` system calls. We build on this interface to save the socket parameters during checkpoint and restore it during restart. For correctness, the entire set of the parameters is included in the saved state (for a comprehensive list of such options, refer to [33]).

The socket's receive and send queues are stored in the kernel. They hold intermediate data that has been received by the network layer but not yet delivered to (read by) the application, as well as data issued by the application that has not yet been transmitted over the network.

With unreliable protocols, it is normally not required to save the state of the queue. Packet loss is an expected behavior and should be accounted for by the application. If the restart does not restore a specific segment of data it can be interpreted as a legitimate packet loss. One exception, however, is if the application has already "peeked" at (that is, examined but not consumed) the receive queue. This is a standard feature in most operating system and is regularly used. To preserve the expected semantics, the data in the queue must be restored upon restart, since its existence is already part of the application's state. With reliable protocols, on the other hand, the queues are clearly an integral part of the socket state and cannot be dispensed of. Consequently we chose to have our scheme always save the data in the queues, regardless of the protocol in question. The advantage is that it prevents causing artificial packets loss that would otherwise slowdown the application shortly after its restart, the amount of time it lingers until it detects the loss and fixes it by retransmission.

In both cases (reliable and unreliable protocols) in-flight data can be safely ignored. Such data will either be dropped (for incoming packets) or blocked (for outgoing packets) by the network layer, since the pod's network is blocked for the duration of the checkpoint. With unreliable protocol this is obviously an expected behavior. Reliable protocols will eventually detect the loss of the data and consequently retransmit it.

Saving the state of the receive queue and the send queue necessitates a method to obtain their contents. It is critical that the method be transparent and not entail any side-effects that may alter the contents of the queue. Should the state of the queue be altered, it would be impossible to perform error recovery in case the checkpoint operation is to be rolled back due to an error in a posterior stage. Moreover, if the intent is to simply take a snapshot of the system, a destructive method is entirely inadequate as it will adversely affect the application's execution after the snapshot is taken.

One method to obtain the contents of the receive queue is to use the `read` system call in a similar way as applications, leveraging the native kernel interface to read directly from a socket. To avoid altering the contents of the queue by draining the data off, this can be done in "peek" mode, that only examines the data but does not drain the queue. Unfortunately, this technique is incomplete and will fail to capture all of the data in the network queues with TCP, including crucial out-of-band, urgent, and backlog queue data.

Another approach is to examine the socket directly and read the relevant data by traversing the socket buffers at a low level. However, the receive queue is of asynchronous nature and is tightly integrated with the implementation of the TCP/IP protocol stack. Reading the chain of buffers requires deep understanding of the relevant kernel mechanisms as well as interpretation of protocol specific information. The result is a prohibitively intricate and non-portable approach.

To get around this we adopt the approach described below, that handles the restore of a socket's receive queue. In particular, we read the data off the socket using the standard `read` system call, while at the same time injecting it back into the socket. The data ends up attached to the socket as if it has just been restored. Effectively this means that even though the receive queue was modified, the application is still guaranteed to read this data prior to any new data arriving on the network, similar to other restored data.

The kernel does not provide interface to insert data into the receive queue, and doing so requires intimate knowledge of the underlying network protocol. This difficulty is overcome by observing that it is sufficient that the application consumes the restart data before any newer data that arrives to the socket. We therefore allocate an alternate receive queue in which this data is deposited. We then interpose on the socket interface calls to ensure that future application requests will be satisfied with this data first, before access is made to the main receive queue. Clearly, the checkpoint procedure must save the state of the alternate queue, if applicable (e.g. if a second checkpoint is taken before the application reads its pending data).

Technically, interposition is realized by altering the socket's dispatch vector. The dispatch vector determines which kernel function is called for each application interface invocation (e.g. `open`, `write`, `read` and so on).

Specifically we interpose on the three methods that may involve the data in the receive queue: `recvmsg`, `poll` and `release`. Interposition only persists as long as the alternate queue contains data; when the data becomes depleted, the original methods are reinstalled to avoid incurring overhead for regular socket operation.

Interposing on `recvmsg` is required in order to use the alternate queue as the source for the data, rather than the original queue. The `poll` method is included since it provides asynchronous notification and probing functionality by examination of the receive queue. Finally, the `release` method is important to properly handle cleanup (in case the data has not been entirely consumed before the process terminates).

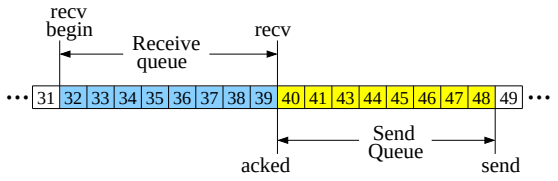
Extracting the contents of the send queue is more involved than the receive queue, as there does not exist a standard interface from which we can leverage, that provides access to that data. Instead the data is accessed by inspecting the socket's send queue using standard in-kernel interface to the socket layer (normally used by protocol code and device drivers). This is accomplished without altering the state of the send queue itself. While the receive queue is tightly coupled to the protocol specifics and roughly reflects the random manner in which the packets arrived, the send queue is more well organized according to the sequence of data send operations issued by the application. For this reason, unlike with the receive queue, reading the contents of the send queue directly from the socket buffers remains relatively a simple and portable operation.

Finally, restoring the state of the send queue is almost trivial: given the re-established connection, the data is resent by means of the standard `write` system call. The underlying network layer will take care of delivering the data safely to the peer socket. In the case of migration, a clever optimization is to redirect the contents of the send queue to the receiving pod and merge it with (or append to) the peer's stream of checkpoint data. Later during restart, the data will be concatenated to the alternate receive queue (of course, only after the latter has been restored). This will eliminate the need to transmit the data twice over the network: once when migrating the original pod, and then again when the send queue is processed after the pod resumes execution. Instead it will merge both into a single transfer, from the source pod to the destination pod.

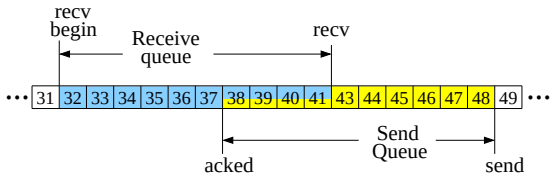
We now discuss how the protocol specific state is saved and restored. The portion of this state that records the protocol properties is exported to the socket layer and can be accessed by the applications. TCP options that activate and deactivate keep-alive timers (`TCP_KEEPALIVE`), and semantics of urgent data interpretation (`TCP_STDURG`) are two such examples. The saved state includes the entire set of these options, and they are handled in a similar way to the socket options, as discussed before. ([33] contains a representative list of such options).

The remaining of the state is internal, and holds dynamic operational data. Unlike accessing the socket layer which is a common practice in kernel modules, access to the protocol's state requires intimate knowledge of its internals. Restoring such a state entails carefully handcrafted imitation of the protocol's behavior. If a portable solution is sought, it is notably desirable to identify the minimal state that must be extracted. As discussed before, the minimal state for unreliable protocols is nil, inherently to their nature. We now discuss reliable protocols.

With reliable protocols, the internal state typically keeps track of the dynamics of the connection to guarantee delivery of messages. Data elements are tagged with a sequence numbers, and the protocol records the sequence number of data that has been transmitted but not yet acknowledged. Timers are deployed to trigger resubmission of unacknowledged data (on the presumption that it had been lost), and to detect broken connections. Each peer in a connection tracks three sequence numbers: last data sent (`sent`), last data received (`recv`) and last data acknowledged by the other peer (`acked`).



(A) Non overlapping queues



(B) Overlapping queues

Figure 4. Non-overlapping and overlapping data queues

An important property of reliable protocols is the following invariant: $\text{recv}_1 \geq \text{acked}_2$ (where the subindices 1 and 2 designate the peers of connection). The reason is that upon receiving data, a peer updates its `recv` value, and sends an acknowledgment. If the acknowledgment is not lost, it will arrive with some small delay, and then the other peer will update its `acked` value. It follows that a send queue always holds data between `acked` and `sent`—that is the unacknowledged data. The receive queue holds data from some point back in time until `recv`. If $\text{recv}_1 > \text{acked}_2$ there will be some overlap between the two queues. This settings is

depicted in Figure 4. The overlap must be fixed during the restart operation, before the application consumes duplicate data. This can be done by discarding extraneous data from either queue. It is more advantageous to discard that of the send queue to avoid transferring it over the network.

We claim that a necessary and sufficient condition to ensure correct restart of a connection, is to capture `recv` and `acked` values on both peers. This data, along with additional protocol specific information, is located in a protocol-control-block (PCB) data structure that is associated with every TCP socket. The concept of a PCB is ubiquitous to the TCP stack, although the details of its layout differ between distinct implementations. It follows that the need to access these fields does not impair the portability of our scheme, but merely requires a trivial adjustment per implementation.

We now show that the minimal state that we need to extract sums up to the aforementioned sequence numbers. Given the discussion above, these values are necessary in order to calculate the extent of the redundant data to be discarded. They are sufficient since the remainder of the data is in the socket queues, and is already handled as described above. It follows that our approach results in a network state checkpoint/restart solution that is almost entirely independent of transport layer protocol. It is optimal in the sense that it requires no state from unreliable protocols, and the minimal state from transport protocols - that portion of the state the reflects the overlap between a send queue and the corresponding receive queue.

Some applications employ timeout mechanism on top of the native protocol, as a common technique to detect soft faults and dead locks, or to expire idle connections. It is also used to implement reliable protocols on top of unreliable ones (e.g. over UDP). The application typically maintains a time-stamp for each connection, updating its value whenever there is activity involving the connection. Time-stamps are inspected periodically, and the appropriate action is triggered if the value is older than a predefined threshold.

It follows that if there is sufficient delay between the checkpoint and the restart, certain applications may experience undesired effect if the timer value exceeds the threshold and expires. We resolve this by virtualizing those system calls that report time. During restart we compute the delta between the current time and the current time as recorded during checkpoint. Responses to subsequent inquiries of the time are then biased by that delay. Standard operating system timers owned by the application are also virtualized. At restart, their expiry time is set in a similar manner by calculating the delta between the original clock and the current one. We note that this sort of virtualization is optional, and can be turned on or off per application as necessary (so that application that strictly require knowledge of the absolute time can operate normally).

ZapC can transparently checkpoint-restart the network state of TCP, UDP and IP protocols, and therefore any distributed application that leverages these widely-used protocols. However, some high performance clusters employ MPI implementations based on specialized high-speed networks where it is typical for the applications to bypass the operating system kernel and directly access the actual device using a dedicated communication library. Myrinet combined with the GM library [1] is one such example. The ZapC approach can be extended to work in such environments if two key requirements are met. First, the library must be decoupled from the device driver instance, by virtualizing the relevant interface (e.g. interposing on the `ioctl` system call and device-dependent memory mapping). Second, there must be some method to extract the state kept by the device driver, as well as reinstate this state on another such device driver.

6 Experimental Results

We have implemented a ZapC prototype as a Linux kernel module and associated user-level tools. Our prototype runs on multiple Linux operating system versions, including the Linux 2.4 and 2.6 series kernels which represent the two major versions of Linux still in use today. We present some experimental results on various size clusters with both uniprocessor and multiprocessor nodes running real applications to quantify ZapC's virtualization overhead, checkpoint and restart latencies, and the resulting checkpoint image sizes.

To measure ZapC performance, we used four distributed applications that use MPI (version MPICH-2 [18]) and PVM (version 3.4), representing a range of different communication and computational requirements typical of scientific applications. One of the applications used PVM while the rest used MPI. Each pod is seen as an individual node so each pod runs one of the respective daemons (mpd or pvmd). The configuration detail needed to do this for more than one pod on a physical machine, is to specify a ssh port default for each pod in .ssh/config that the daemon will use to initiate contact with other pods. The applications tested were:

1. *CPI* — parallel calculation of Pi provided with the MPICH-2 library that uses basic MPI primitives and is mostly computationally bound.
2. *BT/NAS* [8] — the Block-Tridiagonal systems (BT) from the NAS parallel benchmark that involves substantial network communication along the computation.
3. *PETSc* [9] — a scalable package of PDE solvers that is commonly used by large-scale applications, in particular the Bratu (SFI - solid fuel ignition) example, that

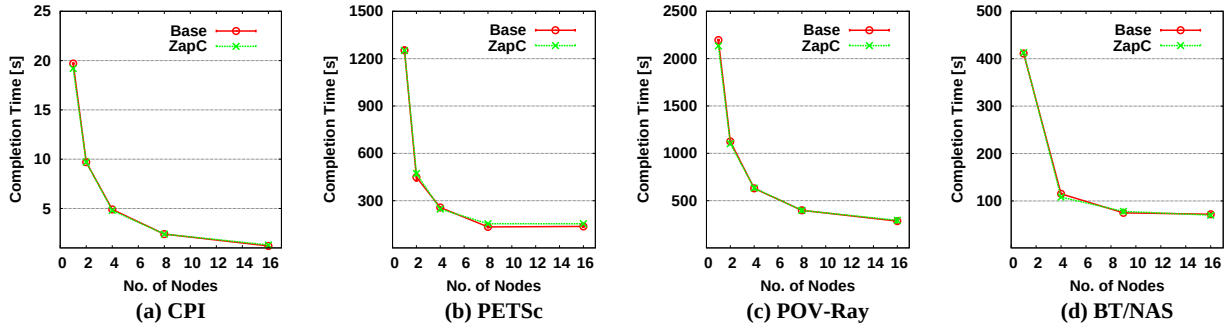


Figure 5. Application completion times on vanilla Linux and ZapC

uses distributed arrays to partition the problem grid with a moderate level of communication.

4. *POV-Ray* [4] (PVM version) — a CPU-intensive ray-tracing application that fully exploits cluster parallelism to render three-dimensional graphics.

The measurements were conducted on an IBM HS20 eServer BladeCenter, a modest-sized cluster with ten blades available. Each blade had dual 3.06 GHz Intel Xeon^{TM} CPUs, 2.5 GB RAM, a 40 GB local disk, and Q-Logic Fibre Channel 2312 host bus adapters. The blades were interconnected with a Gigabit Ethernet switch and connected through Fibre Channel to an IBM FastT500 SAN controller with an Exp500 storage unit with ten 70 GB IBM Fibre hard drives. Each blade used the GFS cluster file system [32] to access the shared SAN.

We measured the applications running across a range of cluster configurations. We ran the ZapC Manager on one of the blades and used the remaining nine blades as cluster nodes for running the applications. We configured each cluster node as a uniprocessor node and ran each application except BT/NAS on 1, 2, 4, 8 and 16 nodes. We ran BT/NAS on 1, 4, 9 and 16 nodes because it required a square number of nodes to execute. We also configured each cluster node as a dual-processor node and ran each of the applications on eight of the nodes. Since each processor was effectively treated as a separate node, we refer to this as the sixteen node configuration. Results are presented with each blade running Linux 2.6, specifically Debian Linux with a 2.6.8.1 kernel. Linux 2.4 results were similar and are omitted due to space constraints.

6.1 Virtualization Measurements

We measured ZapC virtualization overhead by comparing the completion times of the applications running on two configurations, which we refer to as *Base* and *ZapC*. *Base* was running each application using vanilla Linux without ZapC, thereby measuring baseline system performance. *ZapC* was running each application inside ZapC pods. Each execution was repeated five times and the results were averaged over these runs.

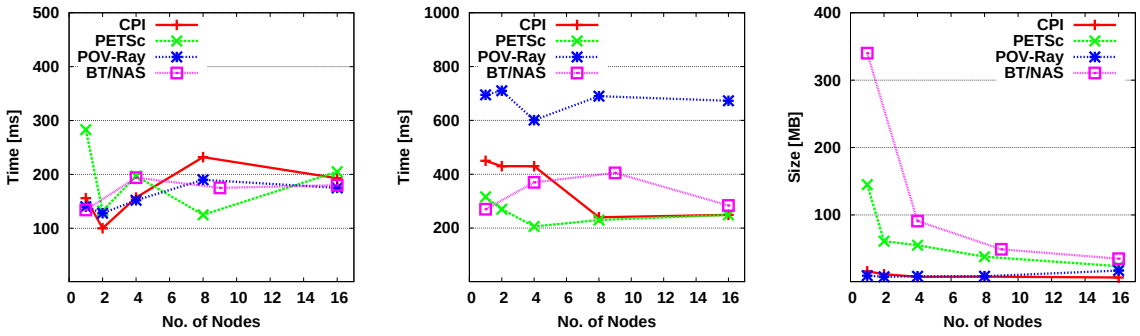
Figure 5 shows the average completion times of the different benchmarks for different numbers of nodes. The results show that the completion times using ZapC are almost indistinguishable from those using vanilla Linux. Results for larger cluster systems were not available due to the limited hardware that was available at the time of the experiments. However, the results on a modest cluster size show that ZapC does not impact the performance scalability of the applications as the relative speedup of the applications running on larger clusters is essentially the same for both vanilla Linux and ZapC.

ZapC virtualization overhead was in all cases much smaller than even the variation in completion times for each configuration across different runs. This further validates that ZapC's thin virtualization layer imposes negligible runtime overhead on real applications. The standard deviation in the completion times for each configuration was generally small, but increased with the number of nodes up to roughly 5%. The variations were largely due to differences in how much work the applications allocated to each node from one execution to another.

6.2 Checkpoint-Restart Measurements

We measured ZapC checkpoint-restart performance by running each of the four distributed applications and taking ten checkpoints evenly distributed during each application execution. We measured checkpoint and restart times for each application as well as the checkpoint image sizes for each application. Due to the limited number of nodes available, restarts were done using the same set of blades on which the checkpoints were performed. These results are shown in Figure 6.

Figure 6a shows the average checkpoint times for each application as measured by taking the average time across all ten checkpoints. This is the time from the invocation of



(a) Checkpoint time **(b) Restart time** **(c) Checkpoint size**

Figure 6. Application checkpoint-restart measurements using ZapC

the Manager by the user until all pods have reported "done", as measured by the Manager. The time includes the time to write the checkpoint image of each pod to memory and represents the time that the application needs to be stopped for the checkpoint to occur, which provides an indication of how frequently checkpoints can be done with minimal impact on application completion time. This does not include the time to flush the checkpoint image to disk, which can be done after the application resumes execution and is largely dependent on the bandwidth available to secondary storage.

Figure 6a shows that checkpoint times are all subsecond, ranging between 100 ms and 300 ms across all four applications. For a given application and a given cluster size, the standard deviation in the checkpoint times ranged from 10% to 60% of the average checkpoint time. For all applications, the maximum checkpoint time was no more than 200 ms of the respective average checkpoint time. For all checkpoints, the time due to checkpointing the network state as seen by the respective Agent was less than 10 ms, which was only 3% to 10% of the average checkpoint time for any application. The small network-state checkpoint time supports the motivation for saving the network state as the first step of the distributed checkpoint, as discussed in Section 4.

Figure 6b shows the restart times for each application as measured based on the time it took to restart the respective application from its checkpoint image. This is the time from the invocation of the Manager by the user until all pods have reported "done", as measured by the Manager. The time assumes that the checkpoint image has already been preloaded into memory and does not include the time to read the image directly from secondary storage. To provide a conservative measure of restart time, we restarted from a checkpoint image taken in the middle of the respective application's execution during which the most extensive application processing is taking place.

Figure 6b shows that restart times are all subsecond, ranging between 200 ms and 700 ms across all four applications. The restart times are longer than the checkpoint times, particularly POV-Ray which takes the longest time to restart. Restart times are longer than checkpoint times in part because additional work is required to reconstruct the network connections of the participating pods. The network-state restart time in most cases ranges between 10 ms and 200 ms.

Figure 6c shows the size of the checkpoint data for each application running with different numbers of cluster nodes. For a given application and cluster configuration, the checkpoint image size shown is the average over all the ten checkpoints of the *largest* image size among all participating pods. Since the checkpoints of each pod largely proceed in parallel, the checkpoint size of the largest pod provides a more useful measure than the total checkpoint size across all pods. In most cases, the checkpoint size of the largest pod decreases as the number of nodes increases since the workload assigned to each node also decreases. The checkpoint size for CPI goes from 16 MB on 1 node to 7 MB on 16 nodes, the checkpoint size for PETSc goes from 145 MB on 1 node to 24 MB on 16 nodes, and the checkpoint size for BT/NAS goes from 340 MB to 35 MB on 16 nodes, an order of magnitude decrease. Only POV-Ray has a relatively constant checkpoint size of roughly 10 MB. These results suggest that the maximum pod checkpoint image size scales down effectively as the size of the cluster increases. This provides good performance scalability for larger clusters since checkpoint size can be an important factor in the time it takes to read and write the checkpoint image to disk.

For the applications we measured, the checkpoint size of the largest pod was much larger than the portion of the checkpoint size due to network-state data. The size of the network-state data was only a few kilobytes for all of the applications. For instance in the case of CPI, the network-state data saved as part of the checkpoint ranged from 216 bytes to 2 KB. Most parallel applications are designed to spend significantly more time computing than communicating given that communication costs are usually much higher than computation costs. Therefore, at any particular point

in time, it is most likely that an application has no pending data in its network queues as it is likely to have already been delivered. It follows that the application data largely dominates the total checkpoint data size. Our results show that application data in a checkpoint image can be many orders of magnitude more than the network data.

7 Conclusions

We have designed, implemented and evaluated ZapC, a system that provides transparent coordinated checkpoint-restart for distributed applications on commodity clusters. ZapC achieves this functionality without modifying, recompiling, or relinking applications, libraries, operating system kernels, or network protocols. ZapC provides three key mechanisms. First, it utilizes a thin virtualization layer to decouple applications from the underlying operating system instances, enabling them to migrate across different clusters while utilizing available commodity multiprocessor operating system services. Second, it introduces a coordinated, parallel checkpoint-restart mechanism that minimizes synchronization requirements among different cluster nodes to efficiently perform checkpoint and restart operations. Third, it integrates a network state checkpoint-restart mechanism that leverages standard operating system interfaces as much as possible to uniquely support complete checkpoint-restart of network state in a transport protocol independent manner, for both reliable and unreliable protocols.

We have implemented ZapC across multiple versions of Linux and evaluated its performance on a broad range of distributed scientific applications. Our results demonstrate that ZapC incurs negligible overhead and does not impact application scalability. Furthermore, they show that ZapC can provide fast, subsecond checkpoint and restart times of distributed applications. Our results show that network-state checkpoint and restart account for a very small part of the overall time, suggesting that our approach can scale to many nodes.

8 Acknowledgments

Yuly Finkelberg made many contributions to the ZapC Linux implementation. This work was supported in part by a DOE Early Career Award, NSF grants CNS-0426623 and ANI-0240525, and an IBM SUR Award.

References

[1] Myrinet index page. `http://www.myri.com/myrinet`.

[2] NetApp Web Page. `http://www.netapp.com`.

[3] Netfilter Web Page. `http:/www.netfilter.org`.

[4] POV-Ray Web Page. `http://www.povray.org`.

[5] *MPI: The Complete Reference (Vol. 1) - 2nd Edition*. MIT Press, 1988.

[6] *PVM: Parallel Virtual Machine*. MIT Press, 2002.

[7] A. Agbaria and R. Friedman. Starfish: Fault-tolerant Dynamic Programs on Clusters of Workstations. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 31. IEEE, 1999.

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[9] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc 2.0 User's Manual. Technical Report ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, 1998.

[10] R. A. Baratto, S. Potter, G. Su, and J. Nieh. MobiDesk: Mobile Virtual Desktop Computing. In *Proceeding of the Tenth Annual ACM International Conference on Mobile Computing and Networking (MobiCom 2004)*, pages 1–15, Philadelphia, PA, Oct. 2004.

[11] A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, June 1997.

[12] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. C^3: A System for Automating Application-Level Checkpointing of MPI Programs. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computers (LCPC'03)*, Oct. 2003.

[13] J. Casas, D. L. Clark, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole. MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2):171–216, 1995.

[14] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[15] Y. Chen, J. S. Plank, and K. Li. CLIP - A Checkpointing Tool for Message-Passing Parallel Programs. In *Proceedings of the Supercomputing*, San Jose, California, Nov. 1997.

[16] J. Duell, P. Hargrove, and E. Roman. The Design and Implemenation of Berkeley Lab's Linux Checkpoint/Restart. Technical report, Berkeley Lab, 2002. Publication LBNL-54941.

[17] G. E. Fagg and J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *LNCS: Proceedings of the 7th European PVM/MPI User's Group Meeting*, volume 1908, pages 346–353. Springer-Verlag, 2000.

[18] W. Gropp. MPICH2: A New Start for MPI Implementations. In *LNCS: Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2474, page 7, London, UK, 2002. Springer-Verlag.

[19] P. Gupta, H. Krishnan, C. P. Wright, J. Dave, and E. Zadok. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-1, Computer Science Dept., Stony Brook University, Jan. 2004.

[20] Y. Huang, C. Kintala, and Y.-M. Wang. Software Tools and Libraries for Fault Rolerance. *IEEE Tecnnical Committee on*

Operating System and Application Environments, 7(4):5–9, 1995.

[21] J. Janakiraman, J. R. Santos, D. Subhraveti, and Y. Turner. Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operting Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'05)*, Yokohama, Japan, June 2005. IEEE.

[22] B. A. Kingsbury and J. T. Kline. Job and Process Recovery in a UNIX-based Operating System. In *Conference Proceedings, Usenix Winter 1989 Technical Conference*, pages 355–364, Jan. 1989.

[23] C. R. Landau. The Checkpoint Mechanism in KeyKOS. In IEEE, editor, *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 86–91, Sept. 1992.

[24] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec. 2002.

[25] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, Knoxville, TN 37996, July 1997.

[26] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Conf. Proceedings, Usenix Winter 1995 Technical Conference*, pages 213–223, Jan. 1995.

[27] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. In *Proceedings of the IPPS Second Workshop on Job Scheduling Strategies for Parallel Processing*, Honolulu, Hawaii, Apr. 1996.

[28] E. Roman. A Survey of Checkpoint/Restart Implementations. Technical report, Berkeley Lab, 2002. Publication LBNL-54942.

[29] J. C. Sancho, F. Petrini, K. Davis, and R. Gioiosa. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, Colorado, Apr. 2005.

[30] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Deurll, P. Hargrove, and E. Roman. LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In *Proceedings of the LACSI Symposium*, Santa Fe, New Mexico, Oct. 2003.

[31] B. K. Schmidt. *Supporting Ubiquitous Computing with Statelss Consoles and Computation Caches*. PhD thesis, CS Department, Stanford University, 2000.

[32] S. R. Soltis, T. M. Ruwart, and M. T. O'Keefe. The Global File System. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages 319–342, College Park, MD, 1996. IEEE Computer Society Press.

[33] W. R. Stevens. *Unix Network Programming*, volume 1. Prentice Hall, NJ, USA, 2nd edition, 1998.

[34] G. Su. *MOVE: Mobility with Persistent Network Connections*. PhD thesis, Department of Computer Science, Columbia University, Oct. 2004.

[35] T. Takahashi, S. Sumimoto, A. Hori, and Y. Ishikawa. PM2: High Performance Communication Middleware for Heterogeneous Network Environments. In *Processings of the IEEE/ACM SC2000 Conference*, Dallas, Texas, Nov. 2000.

[36] T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. *Dr. Dobb's Journal*, (227):40–48, Feb. 1995.

[37] H. Zhong and J. Nieh. CRAK: Linux Checkpoint/Restart as a Kernel Module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, New York, Nov. 2001. Publication LBNL-54942.