Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems

G. (John) Janakiraman, Jose Renato Santos, Dinesh Subhraveti[1], Yoshio Turner
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2005-66
April 7, 2005*

E-mail: {john.janakiraman,joserenato.santos,yoshio.turner}@hp.com

checkpointing, snapshot, process migration, error recovery, availability

We present a new distributed checkpoint-restart mechanism, Cruz, that works without requiring application, library, or base kernel modifications. This mechanism provides comprehensive support for checkpointing and restoring application state, both at user level and within the OS. Our implementation builds on Zap, a process migration mechanism, implemented as a Linux kernel module, which operates by interposing a thin layer between applications and the OS. In particular, we enable support for networked applications by adding migratable IP and MAC addresses, and checkpoint-restart of socket buffer state, socket options, and TCP state. We leverage this capability to devise a novel method for coordinated checkpoint-restart that is simpler than prior approaches. For instance, it eliminates the need to flush communication channels by exploiting the packet re-transmission behavior of TCP and existing OS support for packet filtering. Our experiments show that the overhead of coordinating checkpoint-restart is negligible, demonstrating the scalability of this approach.

* Internal Accession Date Only
[1]Currently at Meiosys
To be published in and presented at The International Conference on Dependable Systems and Networks (DSN-2005), 28 June -1 July 2005, Yokohama, Japan Approved for External Publication

Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems

G. (John) Janakiraman Jose Renato Santos Dinesh Subhraveti* Yoshio Turner
Hewlett-Packard Laboratories
{john.janakiraman,joserenato.santos,yoshio.turner}@hp.com

Abstract

We present a new distributed checkpoint-restart mechanism, Cruz, that works without requiring application, library, or base kernel modifications. This mechanism provides comprehensive support for checkpointing and restoring application state, both at user level and within the OS. Our implementation builds on Zap, a process migration mechanism, implemented as a Linux kernel module, which operates by interposing a thin layer between applications and the OS. In particular, we enable support for networked applications by adding migratable IP and MAC addresses, and checkpoint-restart of socket buffer state, socket options, and TCP state. We leverage this capability to devise a novel method for coordinated checkpoint-restart that is simpler than prior approaches. For instance, it eliminates the need to flush communication channels by exploiting the packet re-transmission behavior of TCP and existing OS support for packet filtering. Our experiments show that the overhead of coordinating checkpoint-restart is negligible, demonstrating the scalability of this approach.

1. Introduction

Application checkpoint-restart mechanisms have the potential to significantly improve the operation of computing environments. Application checkpoint-restart can improve fault tolerance by allowing applications to recover from a component failure by restarting from a recent point in the execution. Application checkpoint-restart can also be used to reduce application downtime during hardware and operating system (OS) maintenance by migrating the application to a different machine before the maintenance operation. Finally, application checkpoint-restart can be used to suspend or migrate jobs to support resource management in emerging Utility Computing and Grid environments [6].

An ideal checkpoint-restart mechanism should be general-purpose and application-transparent, supporting a broad class of real applications without requiring application modifications. It must checkpoint and restore all application execution state including user-level state and the state of OS resources used by the application. The mechanism must extend to parallel applications by supporting consistent checkpoint-restart of processes running on multiple machines. Finally, the checkpoint-restart mechanism must have a practical implementation that does not involve designing special OSes or extensively re-engineering standard OSes. Such approaches are impractical because special OSes have only limited appeal, and extensive OS modifications require prohibitive code maintenance overhead.

Several application checkpoint-restart mechanisms are described in the literature, but none of them meet these requirements [13][9][3][12][1][17][15]. Library-based implementations [13][9] require modification of application source code or re-linking of object code with special libraries, and they do not support applications that use system services such as multithreading, interprocess communication, and network sockets. Vendors including SGI, Cray, and IBM have integrated application checkpoint-restart into proprietary systems but their implementation details are not described in public documents. BLCR [3] is a kernel-module based mechanism that requires application modifications in cases where the application uses uncheckpointed resources such as network sockets. Zap [12] is a kernel-module based process migration mechanism that does not require application or base kernel modification. It integrates a checkpoint-restart mechanism but this mechanism cannot checkpoint and restore network socket state fully. However, as we show in the rest of this paper, the Zap [12] architecture can be extended to realize a checkpoint-restart mechanism that meets our requirements. Systems for checkpointing parallel applications have been built using single node checkpoint-restart mechanisms (e.g., MPVM [1] and CoCheck [17] using Condor's checkpointing [9] and LAM-MPI [15] using BLCR [3]) but they are only applicable for applications using specific message-passing libraries.

In this paper, we describe Cruz, a general-purpose, application-transparent checkpoint-restart mechanism. Cruz's single-node checkpoint-restart mechanism builds on the Zap [12] architecture, thus avoiding the need for

*Currently at Meiosys

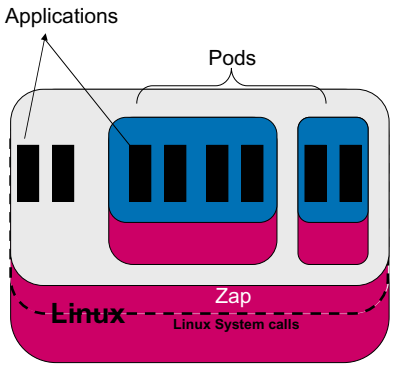


Figure 1. Zap architecture

application or base kernel modification. We have enhanced Zap in many ways enabling it to checkpoint and restart complex applications such as databases and MPI applications. We focus our discussion in this paper on one critical capability, the ability to migrate network addresses and save and restore live network socket state in a manner that is transparent to the application and external clients. To our knowledge, this capability has not been reported in previous literature. Leveraging our ability to capture a snapshot of the network socket state of processes on a single machine, we devise a novel protocol and method for implementing coordinated checkpoint-restart of a distributed set of processes. Our new method eliminates the expensive step of flushing communication channels required in prior approaches for coordinated checkpoint-restart [1][17][15]. Our approach applies generally to TCP/IP [18] based applications and does not require modifications to MPI [16] or PVM [7] implementations.

The rest of the paper is organized as follows. Section 2 reviews related work and presents an introduction to the Zap [12] architecture. Section 3 gives an overview of our solution approach for providing general-purpose application checkpoint-restart. Section 4 describes our extensions to Zap which enable improved support for checkpoint-restart of networked applications. Section 5 describes our solution for global coordinated checkpoint-restart of distributed applications. We present results from performance evaluations in Section 6 and conclude in Section 7.

2. Related Work

Checkpoint-restart mechanisms have been implemented at user-level and at kernel-level. User-level library-based implementations [9][13] lack support for saving/restoring kernel state other than open files and they require application modifications or re-linking. Thus they work only for a narrow set of applications. Kernel-level checkpoint-restart mechanisms integrated into OSes such as SGI's IRIX and Cray's Unicos are application-transparent and provide support for saving/restoring substantial portions of the application's kernel state including process IDs, shared memory, and pipes. However, they do not save/restore network socket state and cannot restart applications when saved process IDs/process group IDs are assigned to other running applications. Moreover, extensively re-engineering the OS to integrate support for checkpoint-restart is not practical for most mainstream OSes.

BLCR [3] and Zap [12] implement checkpoint-restart for Linux at the kernel level in an alternative way, using dynamically loaded kernel modules without modifying the base kernel source code or requiring its re-compilation. BLCR [3] saves/restores process state using a pre-existing kernel module, VMADump[1], which was designed to implement remote process forking in Beowulf clusters. BLCR has extended VMADump to support multithreaded processes and plans to add support for files, pipes, and other features. However, BLCR does not support the checkpoint-restart of communication socket state or the preservation of application's IP addresses across checkpoint-restart. Instead, BLCR [3] relies on applications or message passing libraries to be modified to work around these issues. Also, BLCR cannot restart an application successfully if any of its process IDs are already in use during restore.

Zap [12] is an application-transparent process migration system for Linux implemented as a kernel module. Zap, illustrated in Figure 1, has two components: a thin virtualization layer to create secure environments called PrOcess Domains or *pods* and a mechanism for checkpointing and restarting pods. The virtualization layer intercepts system calls to expose only virtual identifiers (e.g., virtual process IDs) to the processes within a pod instead of the physical identifiers returned by the OS. This results in a private name space for each pod which isolates it from other pods and decouples it from the OS. Checkpoint stops all the processes within the pod and extracts their state, including OS resource state relevant to the processes. Restart re-creates these processes and restores their execution state, mostly by invoking system calls. While the re-created OS resources have different identifiers inside the operating system, Zap's virtualization layer masks this difference from applications. Hence, Zap can restart processes successfully even when their process ids are already in use within the operating system, a unique capability that is not provided by BLCR [3] or the kernel-based implementations mentioned previously. Our work uses and builds on the powerful architecture of Zap to realize a general-purpose checkpoint-restart implementation. The original Zap implementation [12] provided support for a broad class of OS resources including: process virtual memory, CPU state, file descriptors, pipes, signals, and terminal state. Zap does not save or restore file system state to avoid the high performance cost of transferring file

[1] http://bproc.sourceforge.net

system state. Like typical checkpoint-restart mechanisms, it relies on a network-accessible file system that is accessible from any machine on which the application may be restarted. Integration of Zap with a file system with snapshot capability would enable checkpoint and restart of both file system and computation state.

We have enhanced the original implementation of Zap by adding the capability to checkpoint and restart OS resources such as shared memory, semaphores, threads, sockets and transient socket buffer state. We also extend Zap's virtualization layer to provide unique, externally routable IP addresses for each pod and preserve them across checkpoint and restart. Thus networked applications can be checkpointed on one machine and restarted on a machine with a different IP address without changing the application's IP address and its connection state from the perspective of the application or its remote clients.

Practical checkpoint-restart mechanisms for distributed applications are limited to applications that communicate using message passing models such as MPI [16] and PVM [7]. MPVM [1], CoCheck [17], and LAM-MPI [15] modify the message passing library (PVM or MPI) to implement coordinated checkpoint and restart. These systems change the library as follows (with differences in details). They flush all the messages that are in flight between the application's processes during checkpoint. They re-establish network connections among the processes at restart. Finally, since processes may be restarted on different machines than they were running on before checkpoint, the libraries are modified to reconstruct location information at restart. In contrast, our coordinated checkpoint-restart approach eliminates all these steps resulting in a much simpler, more efficient, and more scalable implementation. We exploit our capability of saving and restoring TCP socket state and socket buffer state to eliminate the steps of flushing communication channels (instead, we simply drop all in-flight packets) and closing/reopening communication channels. We preserve IP addresses for applications, thereby allowing processes to use normal IP mechanisms to locate each other after restart. With these changes, the number of coordination messages is reduced to the minimum necessary for ensuring the checkpoint is committed on all nodes (i.e., equivalent to a two-phase commit [8]). Furthermore, our resulting coordinated checkpoint-restart mechanism can work for general TCP-based applications (including MPI and PVM applications) without any changes to applications or libraries.

A few coordinated checkpoint mechanisms designed to work in conjunction with message logging mechanisms [4][11] have exploited the message logs to eliminate the need to flush communication channels at checkpoint. Logging messages has prohibitive performance overhead for communication-intensive applications, so it is not a feasible substitute for flushing channels at checkpoint. Furthermore, in contrast to our approach, these techniques still require special communication libraries to log messages, to reestablish network connections at restart, and to deal with address relocation issues.

A few industry vendors offer products that integrate checkpoint-restart functionality. Meiosys[2] offers a MetaCluster product capable of checkpointing and restarting distributed applications. MetaCluster has capabilities similar to our mechanism (e.g., saving and restoring TCP and UDP connections and working without requiring application modifications), but the details of its implementation are not publicly available. VMWare[3] provides virtual machine products including the capability to suspend, restart, and migrate virtual machines between physical machines. VMWare's mechanisms differ from our approach in that they checkpoint and restore the entire operating system including all its applications instead of just a single application, which imposes higher overhead at checkpoint and restart time. VMWare's hardware virtualization also imposes substantial performance overhead at runtime.

3. Solution Overview

In this paper we present Cruz, a powerful and general-purpose checkpoint-restart mechanism. Cruz is application-transparent, has a practical implementation based on a kernel module without requiring modifications to the base kernel, supports application migration between machines without losing network connections, and efficiently supports checkpoint-restart for distributed applications.

To realize these capabilities, we extend Zap to improve its support for migrating networked applications. Our extensions to Zap enable it to save and restore the state of all TCP connections used by a process as part of the process state. This includes any received data that has not yet been delivered to the process, any data submitted by the process which has not yet been successfully acknowledged by the recipient, and the state of the TCP connection. In addition, we extend Zap to support network interface virtualization to enable assigning a unique, externally routable IP address to a pod which is preserved across checkpoint and restart. These extensions enable pods to be migrated within an IP subnet without disrupting communication, even if the remote processes are not controlled by Zap.

The ability to capture TCP connection state enables an elegant solution for the checkpoint-restart of distributed processes that exploits the reliable messaging properties of TCP. At checkpoint, only the states of the individual processes (which includes the state of their TCP connections)

[2] http://www.meiosys.com
[3] http://www.vmware.com

are saved on the respective machines and any in-flight messages in the network are discarded. At recovery, the TCP connection states are restored and the TCP protocol ensures that any messages which were previously discarded are retransmitted. A standard distributed commit protocol is used to ensure atomicity of the checkpoint and restart.

4. Checkpoint-Restart for Networked Applications

This section describes our extensions to Zap's support for migrating networked applications. Section 4.1 explains how Zap is extended to save/restore network socket state including socket send and receive buffer contents. Although the original Zap implementation had support for saving and restoring a portion of socket state, it lacked support for saving/restoring data in send/receive buffers. Section 4.2 describes new support for assigning a network-visible IP address to each pod which is preserved across migration. The original Zap implementation assigned each pod a virtual IP address that was not network-visible and relied on Zap at both ends of the network connection to translate between the virtual IP address and the network-visible IP address of the host (by rewriting packet headers).

4.1. Network State Checkpoint-Restart

An application prepares to establish a TCP [18] connection by invoking the OS to create an object called a socket. The OS maintains the socket's state which includes source/destination IP addresses/ports, TCP connection state including sequence numbers and connection status, and various socket options that affect the data transfer. The socket state also includes the contents of socket send/receive buffers which reside in the kernel and store data that are awaiting network transmission, acknowledgement, or delivery to the application.

To accurately capture the state of all sockets used by the application, the checkpoint-restart mechanism must ensure the application's socket states cannot change during the checkpoint procedure. They cannot be changed by the application processes since Zap sends SIGSTOP signals to stop the execution of all processes in a pod before checkpointing it. However, OS kernel threads and the network device driver could change the socket states. Thus we extend Zap such that at checkpoint time it acquires spin locks that are required by the kernel for network processing to prevent the network stack from delivering or transmitting packets from/to the application's sockets. Since these locks are held only for the duration needed to save the socket states (rather than the entire application state), the kernel's network processing is blocked only for a short duration.

We further extend Zap to checkpoint the frozen socket state as follows. A socket's receive buffer stores application-level byte stream data which has been received from the network but is not yet delivered to the application. To checkpoint this data, we extend the Zap implementation to call the socket receive system call on behalf of the application. Since checkpointing should be a non-destructive operation which allows the checkpointed process to resume execution immediately after the checkpoint operation is complete, the `MSG_PEEK` option is passed to socket receive to read but not remove data from the buffer.

Linux lacks a system call interface that would allow Zap to access the contents of socket send buffers. Therefore, we obtain this data by directly walking the send buffer's kernel data structure. The data packetization that is indicated in the send buffer must be preserved across checkpoint and restart because the Linux TCP stack expects ACK sequence numbers to correspond to packet boundaries. We extend Zap to read and save the application-level data found in the send buffer and record the packet boundaries, which are preserved on restart as described later.

Finally, Zap reads the TCP connection state directly from the socket data structure. As we describe below, when an application is restarted from a checkpoint image, its sockets are re-created with empty buffers before the saved socket buffer data is restored. Therefore, the checkpoint procedure saves a modified version of the TCP connection state which reflects an empty receive buffer in which the current contents have been successfully delivered to the application, and an empty send buffer in which the current contents have not yet been issued by the application to the OS. The necessary modifications to the connection state are minimal and consist of changing the values of two sequence numbers in the saved copy of the socket data structure.

When restarting an application from a saved checkpoint image, Zap creates the number of sockets that were checkpointed. These new sockets are initialized with the saved TCP connection state which indicates empty send and receive buffers. We extend Zap to issue a sequence of socket send system calls to restore the saved sequence of send buffer data blocks. To preserve the original send-side packet boundaries, we issue individual send operations for the data associated with each packet. In addition, the mechanism temporarily sets the socket TCP options to disable the Nagle algorithm and other mechanisms that could change the packet boundaries (e.g., TCP_CORK in Linux) before issuing the send system calls.

It is cumbersome to insert application-level byte stream data into socket receive buffers, which are designed to receive packet data directly from the network. Therefore, we extend Zap to restore this data for each socket by copying it to an alternate buffer which the mechanism allocates for the socket in kernel address space. Zap's system call intercep-

tion mechanism is configured to intercept the socket receive system call such that data stored in the socket's alternate buffer is transparently delivered to the application when it issues a receive call for the socket. The interception code checks if the socket's alternate buffer is empty, and if so invokes the original socket receive system call to deliver data from the socket's receive buffers. As a performance optimization, the interception of the socket read system call is removed when the alternate buffers for all sockets become empty. If a checkpoint is initiated when the alternate buffers are not empty, data in the alternate buffers and any data in the socket receive buffers are both retrieved through the intercepted socket read system call. Data from both buffers are concatenated and saved in the checkpoint. This mechanism allows a checkpointed application to transparently continue network communication with other processes after restart. While network packets can be dropped or be received multiple times across checkpoint and restart, the underlying TCP protocol handles these cases transparently.

We have verified that our current implementation works correctly for multiple kernel versions (e.g., Linux 2.4.20 and Linux 2.4.25). In general, however, the representation of socket data structures in a new kernel version could be different to the extent that our implementation would have to be ported to accommodate the changes. Porting effort can be minimized if OSes can be extended with a small set of new interfaces to provide high-level access to internal network state (e.g., sequence numbers). Such extensions have been proposed previously [10] and we are investigating their feasibility.

4.2. Network Address Migration

We have extended Zap to support persistent, externally routable addresses to pods. This is accomplished by attaching to each pod a *virtual network interface* (VIF) which is the only network interface that is visible to processes within the pod. The VIF can be assigned a network-visible IP address and an ethernet MAC address. When a pod is migrated, its VIF is deleted at the original host and a new VIF is created at the destination host. The new VIF is attached to the migrated pod and is assigned the same addresses as the original VIF. This enables remote processes to continue communicating with a migrated process even if the remote processes are not under control of Zap. Since the IP address assigned to a pod is visible to the network, this approach requires the source and destination of migration to be within the same routing domain (e.g., the same IP subnet). Several operating systems, including Linux, support the creation of VIFs and assignment of IP addresses to VIFs. Our extensions to Zap use this feature to provide VIFs for pods.

A pod's VIF can be assigned a unique static IP address by the system administrator or alternatively it can be assigned a dynamic IP address if a DHCP client process running in the pod queries a DHCP server on the network. The DHCP client composes a query message which contains the MAC address of the pod's VIF. The DHCP server associates the MAC address with an appropriate IP address assignment which it returns to the client. Since both the IP and MAC addresses for a pod are migrated along with the pod, migration is transparent to both the DHCP server and the client.

Since multiple VIFs may share a physical ethernet interface, a pod's VIF can be assigned a unique network-visible MAC address that can be migrated with the pod (as discussed above) only if the ethernet hardware supports multiple MAC addresses or if it can be placed in promiscuous mode. Otherwise, all VIF's that share a physical interface must share its MAC address, and this MAC address cannot be migrated with a pod (since other pods which are not being migrated are using it). We have developed an alternate solution for such environments. With this solution, when a pod migrates, the pod's VIF starts using a different physical interface with a different MAC address even though it keeps the same IP address it had before the migration. The standard Address Resolution Protocol (ARP) is used to update the network about this new mapping of IP address and MAC address. If the IP address is static, this is sufficient. However, for dynamic IP addresses, the DHCP server uses a MAC address specified in the payload of the DHCP request to identify the client and renew its lease for the IP address. Unless this MAC address is preserved across migration, the dynamic IP address assigned to the client will change at the end of the lease causing active network connections to be lost. To avoid this problem, we ensure that the DHCP client uses a *fake MAC address* which is preserved across migration. This is achieved by extending Zap to intercept network device-related `ioctl` calls to provide a virtualized view of network hardware. In particular, the `SIOCGIFHWADDR` request type is intercepted to return the fake MAC address of an interface. The DHCP client invokes this request and embeds the fake MAC address in its DHCP request message.

A primary implementation challenge for our approach is to confine a pod's processes to use only the pod's VIF for accepting incoming network connections and for initiating outgoing network connections. To prepare a socket to listen for incoming connections, a process issues the `bind` system call and gives a local network address as the argument, or else specifies that the socket can bind to any local IP address. To ensure that the calling process can only accept connections that are incoming to the pod's IP address, we extend Zap to intercept the `bind` system call using a simple wrapper function which checks if the calling process is in a pod, and if so replaces the network address argument with the IP address of the pod's VIF.

A process initiates an outgoing network connection by calling the `connect` system call with arguments specify-

ing the remote network address and the socket to connect. The `connect` system call implicitly binds the socket to a local network address (IP address and free IP port) which is chosen by the OS. We extend Zap to intercept the `connect` system call using a simple wrapper which invokes `bind` prior to the original function that implements `connect`. The wrapper ensures that sockets in a pod are bound to the pod's IP address on a free port.

5. Checkpoint-Restart of Distributed Processes

The mechanisms described so far allow application state of processes in a single pod to be checkpointed and restored atomically. For a parallel application with processes running on multiple machines, the checkpoint and restart of the application's processes on each machine must be orchestrated so that the global application state is consistent [2]. For simplicity, in the following description we use the terms "node" and "pod" interchangeably to refer to the set of processes on a machine that are part of the distributed application. Chandy and Lamport [2] have shown that a global checkpoint state is consistent if it satisfies the following properties: 1) if any node's state indicates a message has been received, the sending of the message must be reflected in the state of the sender, and 2) if any node's state indicates a message has been sent but the state of the intended recipient does not indicate that the message has been received and the communication channel is reliable, then the message must be saved as part of the state of the channel between these two nodes. *Coordinated checkpointing* is a well-known technique for consistent checkpointing which we employ in our solution. Pros and cons of coordinated checkpointing and alternative approaches are well documented in the literature so we do not discuss them here [5].

With coordinated checkpointing, nodes checkpoint concurrently and employ a coordination protocol to ensure their checkpoint states are globally consistent. To guarantee the first consistency property, the protocol prevents any message sent after a node has completed its checkpoint from becoming part of the receiver's checkpoint state. To guarantee the second consistency property, the protocol also ensures all messages in transit over the channel are saved (the communication channel is reliable in most environments through the use of TCP). Prior coordinated checkpointing implementations had no means to capture the state of the systems implementing the communication channel (e.g., TCP state, state of network switches). Consequently, their protocols require each node to exchanges markers with every other node to flush in-transit messages to the receiver, where they are saved. Our single node checkpoint-restart mechanism can save and restore the state of TCP which implements the reliable communication channel (Section 4.1). With the TCP state included as part of the checkpoint state, the uncaptured in-transit messages constitute only the state of the unreliable communication channel (state on network switches and routers). Since the state in this unreliable communication channel can be ignored without violating the consistency requirements, we develop a simpler coordination protocol that does not flush in-transit messages.

The steps in our coordinated checkpoint algorithm are specified at a high level and shown by illustration in Fig. 2. When a parallel application must be checkpointed, the Checkpoint Coordinator notifies a Checkpoint Agent on each machine on which the application is running. Each Agent reacts to the notification by disabling all network communication for the local pod that hosts the application[4] (in Linux, for example, the Agent can add a netfilter rule which ensures that all traffic to or from the local pod is silently dropped). This step isolates the local pod's state and prevents it from being changed by pods that compose the distributed application on other machines. The Agent next uses our single-node checkpoint mechanism to save the local pod's state independently. Since the pod's state includes TCP state, any dropped messages will be re-transmitted by TCP when normal execution resumes. When all pods have successfully checkpointed their individual states, the set of saved states constitutes a consistent global state of the system. Once the Checkpoint Coordinator is informed by all Agents that local checkpoints have been successfully completed, it notifies each Agent to allow the pods to resume execution. Each Agent re-enables communication for its local pod (by undoing the netfilter configuration rules, in the case of Linux) and allows the application processes in the local pod to run.

Coordinated restart operates similarly except, of course, state is restored from checkpoint instead of being saved. It is necessary to disable communications as the first step of restart even though application processes have not been restored at that point. This prevents application messages from being sent on the network prematurely before the application state is restored fully on all pods. If communication were not disabled, the OS would resume transmitting messages on the network as soon as the TCP connection state of the pod is restored. These messages may arrive at a pod before the appropriate connection state has been restored because each pod restores its state at a different rate. This would result in an error which would destroy the connection and cause application failure. This situation is prevented by disabling communications as the first step. When all pods have completed restoring their state, each pod can be notified to resume their operation (enable communications and allow processes to run).

The simplicity of our mechanism is readily apparent.

[4] Since the Agent on each machine runs outside of the application's pod, this step does not disrupt communication between the Checkpoint Agent and the Checkpoint Coordinator.

```
Checkpoint Coordinator:
  Step 1: Send <checkpoint> message to all Agents.
  Step 2: Wait to receive <done> from all Agents.
  Step 3: Send <continue> message to all Agents.
  Step 4: Wait to receive <continue-done>
          from all Agents.

Checkpoint Agent:
  (when <checkpoint> message is received)
  {
     Step 1: Configure ipfilter to drop
             packets to/from local pod.
     Step 2: Stop local pod's processes and take
             local checkpoint.
     Step 3: Send <done> to Coordinator.
     Step 4: Wait to receive <continue>.
     Step 5: Resume stopped processes in
             local pod.
     Step 6: Configure ipfilter to allow
             packets to/from local pod.
     Step 7: Send <continue-done> to
             Coordinator.
  }
```

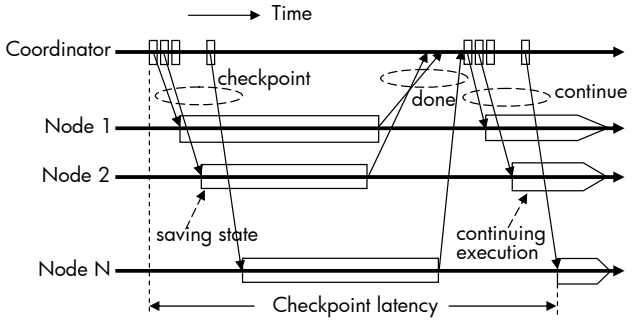


Figure 2. Global Coordinated Checkpoint Algorithm.

Previous mechanisms implement all-to-all communication protocols which either send messages on each pairwise channel to flush all in-transit messages [17][1] or exchange messages between every pair of nodes to estimate the amount of data that must be received before checkpoint can proceed [15]. Our approach eliminates this complexity. The state of communication channels, i.e., packets that are in-flight in the network, is not saved. Instead, packets received from the network after communication has been disabled are silently dropped at the lowest levels of the OS network stack. Since the checkpoint state of each pod includes the state of its TCP connections, any dropped messages will be automatically recovered by TCP's reliable message protocol during normal execution (when the application continues computation after checkpoint or when the application restarts from this checkpoint state). Coordinated restart is similarly simplified over previous implementations, since our approach neither requires the exchange of application-level messages to discover the new locations of processes nor the establishment of new connections between every communicating pair of processes. As we will discuss shortly, the simpler implementation also improves performance and scalability.

The coordination algorithm we have described (Fig. 2) can be extended in a straightforward way to tolerate Coordinator and Agent failures. We have not included these extensions, which are well-known, to highlight the simplicity enabled by the capability to migrate network state.

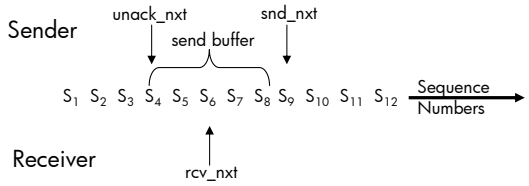


Figure 3. TCP communication of a pair of processes. All pointers advance to the right.

5.1. Correctness Discussion

Our coordinated checkpoint-restart approach leverages the reliable delivery properties of TCP. TCP uses packet sequence numbers, acknowledgements, send buffers[5], and timers to ensure exactly-once and in-order message delivery. Figure 3 shows a simplistic representation of the key elements of TCP state at the sender and receiver. The sender's TCP state maintains two pointers into the sequence number stream: $unack\_nxt$ which is the smallest unacknowledged sequence number and snd_nxt which is the sequence number that will be used for the next packet that is sent. All data packets with sequence number less than $unack\_nxt$ have been successfully acknowledged. Data packets with sequence numbers from $unack\_nxt$ and less than snd_nxt have been sent but are not yet successfully acknowledged. These packets are maintained in the send buffer. The receiver maintains a variable rcv_nxt which is the next sequence number that it expects to receive. Data packets with sequence numbers less than rcv_nxt have been received and an acknowledgement packet has been sent for them. During normal TCP operation the predicate

$$unack\_nxt \leq rcv\_nxt < snd\_nxt \qquad (1)$$

is invariant. This invariant, the send buffers, and TCP's retransmission protocol collectively guarantee that all data will eventually be successfully received by the receiver and successfully acknowledged.

Our mechanism for saving and restoring network state (Section 4.1) saves $unack\_nxt$, snd_nxt, and the send buffers of packets with sequence numbers from $unack\_nxt$ and less than snd_nxt at the sender. This state is saved for every connection but we focus our attention on one of these connections without loss of generality. Our mechanism also saves and restores rcv_nxt at the receiver. However, the sender and receiver states are saved and restored on different physical nodes and thus at different times. If we can

[5]Receive buffers are not central to the reliable properties of TCP

show that the previous invariant is maintained in any global checkpoint state in spite of the asynchrony, that would guarantee that all messages will be successfully delivered during continued operation from this global state (whether the operation continues after the checkpoint completes or after restarting from this saved checkpoint state). This would effectively satisfy the sufficient conditions described in [2] proving the consistency of the checkpoint and restart.

We show that the TCP invariant remains satisfied in any global checkpoint state if the checkpoints of participant nodes are coordinated using the protocol described in Fig. 2 Consider the instant when the Checkpoint Coordinator begins to execute its Step 3. It is straightforward to observe from the coordination protocol that, at that instant, communication is disabled on all nodes and, hence, the TCP state is frozen on all nodes (i.e., the elements of TCP state mentioned above cannot change on any node). If this TCP state on each node is saved, this would amount to capturing the TCP state synchronously on all nodes which would trivially satisfy the TCP invariant. In the Checkpoint Agent's operation described in Fig. 2, the TCP state is actually captured at an earlier time on each node. However, we observe that each node disabled its communication (in Step 1) before capturing its TCP state and this communication is reenabled (in Step 7) well after the instant discussed above. Consequently, the TCP state saved at each node is identical to the TCP state on the node at the instant discussed above. Hence, the global checkpoint state preserves the TCP invariant and is thus consistent.

Our distributed restart mechanism disables communication on each node before restoring its part of the global state. The restart of participant nodes is coordinated using a protocol identical to the coordinated checkpoint protocol. An argument similar to the one above can be used to show that our coordinated restart mechanism resumes the processes and the communication from the consistent global checkpoint state resulting in correct execution.

5.2. Performance Discussion

Our coordinated checkpoint-restart approach is lightweight and scalable, improving significantly over previously proposed mechanisms [1][17][15]. The messages exchanged to coordinate checkpoint and restart in our approach (Fig. 2) are the minimum necessary to ensure the atomicity of the global checkpoint, such as with two-phase or three-phase commit protocols. The approaches used in MPVM [1], CoCheck [17], and LAM-MPI [15] require additional messages to flush the channels between every pair of processes. This results in $O(N^2)$ message complexity compared to $O(N)$ complexity with our approach.

The algorithm we described in Fig. 2 blocks processes until all nodes have completed their checkpoint. Our mechanisms can be extended with optimizations explored in prior research such as copy-on-write to allow applications to compute concurrently with their checkpointing and incremental checkpointing to minimize checkpoint size. In addition, we now describe an optimization that can permit nodes to continue computation without waiting for all nodes to complete their checkpoints. Recall from Section 5.1 that each node saves its state after its communication has been disabled. We observe that the state of each individual node cannot change as long as its communication is disabled. Therefore, once the Coordinator has confirmed that communication is disabled on all nodes, it can permit each node to continue operation as soon as it has completed saving its checkpoint state, without any possibility of changing the checkpoint state of other nodes. With this optimization (illustrated in Fig. 4), each node notifies the coordinator as soon as its communication is disabled without waiting to save its local state. Mechanisms to support further asynchrony among the checkpointing nodes (e.g., using checkpoint sequence numbers) are not justified since applications will be forced to stall when a non-blocked node tries to communicate with a blocked node.

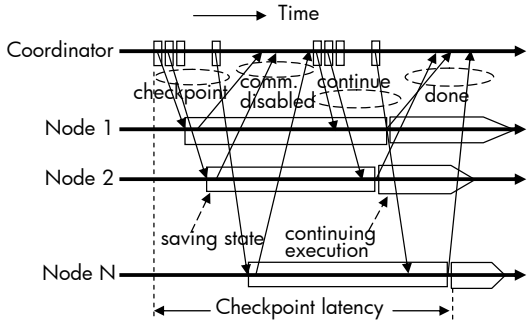
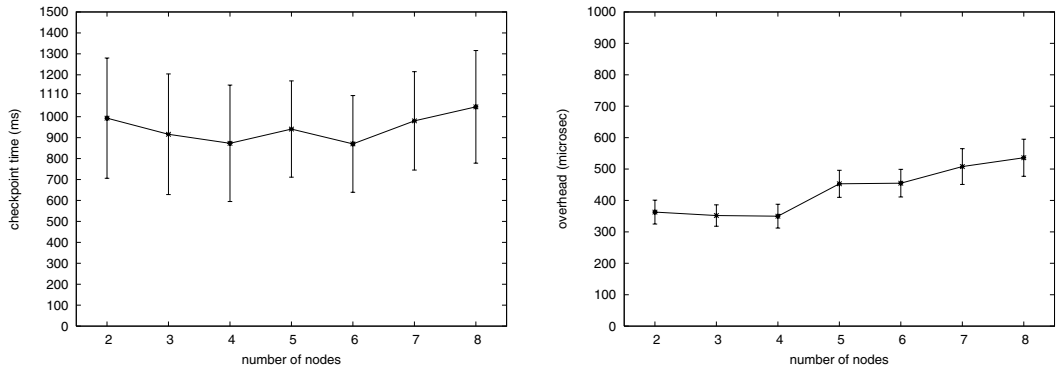


Figure 4. Coordinated Checkpoint Operation with optimization.

Several other optimizations are worth examination. Since communication is disabled at each checkpoint, packets may be lost and TCP may backoff causing performance to be degraded for a short duration after the checkpoint while TCP recovers from its backoff. We evaluate this effect experimentally in Section 6. The impact of TCP backoff can be reduced by keeping communication disabled only for the duration it takes to save the communication state. Since saving the communication state is a fast operation, this allows any recovery from TCP backoffs to proceed in parallel with saving the checkpoint state, which is dominated by the time to save the application's virtual memory state.

6. Performance Evaluation

We have implemented Cruz on a cluster of Linux 2.4 systems and integrated it with LSF [14], a job scheduler for



(a) Total checkpoint latency

(b) Coordination overhead

Figure 5. Results for slm benchmark experiments.

clusters. In this section we show experimental performance results obtained for checkpointing distributed applications with this implementation using two benchmarks: **a)** a semi-lagrangian atmospheric model benchmark (slm), which is a parallel application used for weather prediction; and **b)** a TCP streaming benchmark, consisting of a transmitting node sending data through a TCP socket connection to a receiving node at maximum rate. The benchmarks were executed on a cluster of machines interconnected by a gigabit ethernet switch. Each node consists of two 1 GHz Pentium III processors, 2 GB of RAM, 256 KB of cache and an Intel e1000 gigabit NIC. During all experiments, checkpoint is initiated by a coordinator located on a node distinct from the application nodes.

The runtime overhead of Cruz is negligible (less than 0.5%) since the underlying Zap mechanism requires nothing more than virtualizing identifiers. Figures 5(a) and 5(b) show experimental results for checkpointing the slm benchmark with the number of nodes varying from 2 to 8. The error bars represent the standard deviation of the measurements ($[\mu - \sigma, \mu + \sigma]$) among the total number of measurements observed during one complete execution of the benchmark. During the experiments the application was run from beginning to completion with checkpoints every 8 seconds interval of execution time. The total execution time for the benchmark[6], varies from 545 seconds for 2 nodes to 205 seconds for 8 nodes. Figure 5(a) shows total checkpoint latency, measured in the coordinator, i.e., the time interval elapsed from the first *checkpoint* message sent to the last *done* message received at the coordinator. Figure 5(b) shows the estimated overhead asssociated with coordination. The overhead was computed by subtracting from the total checkpoint latency measured in Figure 5(a), the time spent in executing local operations of *checkpoint* and *continue* in the application nodes. Since application nodes execute these local operations in parallel, we consider the global cost of each local operation as the maximum time measured in all nodes.

The results of Figure 5(a) show an overhead of approximately 1 second for checkpointing the slm benchmark for all node configurations. This time is a function of the size of the application state that needs to be saved, and is dominated by the time to write this state to disk. In general, most of the state consists of the non-zero contents of the virtual memory of all processes running in the pod.

More importantly, the results of Figure 5(b) show that the overhead for coordination is negligible, on the order of 350 μs to 550 μs. The graph shows that the overhead increases by approximately 50 μs for each node for configurations with more than 4 nodes. This suggests that our checkpoint mechanism should scale to a large number of nodes before the overhead becomes comparable with the checkpoint time. Performance results for the restart operation are similar to the results of Figures 5(a) and 5(b) but are omitted here because of space limitations.

We performed experiments using a TCP streaming benchmark running between two nodes to evaluate the performance impact of packet drops in the network when communication is disabled for the nodes to perform checkpoints. Figure 6 shows the measured rate of the TCP stream between two nodes, as a function of time. The plotted rate corresponds to the average rate measured in the receiver during a sliding window of 10 ms duration previous to the corresponding point. A checkpoint operation is started at time $t = 0$ when the rate drops to zero[7]. The checkpoint operation completes after approximately 120 ms. At this time the receiver continues consuming data in the TCP receive buffer that arrived before the checkpoint operation was started, illustrated by the short pulse at 120 ms. How-

[6]not including the time the application is stopped for checkpointing

[7]The curve reaches 0 only at time t=10, since the plotted rate is averaged over the previous 10 ms window.

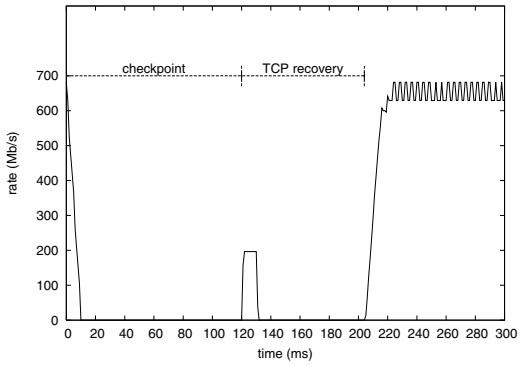


Figure 6. Effect of dropped packets on flow rate (TCP streaming benchmark)

ever, the sender does not continue sending data until later when it recovers from lost packets due to our network filter. At this time communication resumes at the normal rate as before the checkpoint operation. The small oscillations in the curve is due to the fact that the application receives bytes in multiples of packets which in this experiment have the maximum size of 1500 bytes. Although the network filters used in our distributed checkpoint caused packets to be dropped and network communication to be suspended, this perturbation is small, with normal communication restarting after approximately 100 ms.

7. Summary

We presented a powerful and general-purpose checkpoint-restart mechanism, Cruz, which improves the operation of computing environments, reducing both planned and unplanned downtime and increasing resource allocation flexibility. Our mechanism has two main contributions. First, we enable saving and restoring the state of live TCP connections. Second, we leverage this capability and develop a new lightweight distributed checkpoint-restart mechanism which uses the fewest messages necessary to ensure the atomicity of the global coordinated checkpoint. We have implemented our mechanism and evaluated its performance using a scientific parallel application and a network intensive benchmark. Our results show negligible coordination overhead demonstrating the scalability of our approach. The results suggest that the system should scale to a large number of nodes before coordination overhead becomes comparable to the time to perform local checkpoint or restart.

We propose performance optimizations to our base solution. As future work, we plan to evaluate the benefits of these optimizations. In addition, we plan to evaluate performance of our mechanism across a wide range of applications and cluster configurations.

References

[1] 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.

[2] 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.

[3] J. Duell, P. Hargrove, and E. Roman. The design and implementation of Berkeley Labs' Linux checkpoint/restart. 2002.

[4] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Int'l Symp on Fault-Tolerant Computing*, Jun 1994.

[5] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, Sep 2002.

[6] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for distributed system integration. *Computer*, 35(6), 2002.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 2000.

[8] P. Jalote. *Fault Tolerance in Distributed Systems*. PTR Prentice Hall, 1994.

[9] M. Litzkow, T. Tanenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Computer Sciences Technical Report 1346, University of Wisconsin, Madison, WI, 1997.

[10] J. Mogul, L. Brakmo, D. Lowell, D. Subhraveti, and J. Moore. Unveiling the transport. In *Second Workshop on Hot Topics in Networks*, Nov 2003.

[11] N. Neves and W. K. Fuchs. RENEW: A tool for fast and efficient implementation of checkpoint protocols. In *Int'l Symp on Fault-Tolerant Computing*, Jun 1998.

[12] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Fifth Symp. on Operating System Design and Implementation (OSDI 2002)*, pages 361–376, Dec 2002.

[13] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter 1995 Technical Conference*, pages 213–224, 1995.

[14] Platform Computing Corporation. *LSF User's Guide*. `www.platform.com`.

[15] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, October 2003.

[16] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference (Vol. 1) - 2nd Edition*. MIT P ress, 1998.

[17] G. Stellner. Cocheck: Checkpointing and process migration for MPI. In *10th International Parallel Processing Symposium (IPPS 1996)*, 1996.

[18] W. R. Stevens. *TCP/IP Illustrated, Volume 1 The Protocols*. Addison-Wesley, 1994.