A Semantic Checkpointing Framework for Enabling Runtime-reconfigurable Applications

by

Steven Chan

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

© Steven Chan, MMIII. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science

August 11, 2003

Certified by

Karen R. Sollins
Principal Research Scientist
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Abstract
This thesis proposes to enable runtime-reconfigurable applications through the use of semantic checkpointing. We view applications here as a collection of inter-connected components, and reconfigurations as the reconstitution of components that make up an application. By checkpointing only values that are deemed to be of semantic significance, application state is maintained across reconfigurations even if the two configurations have completely different implementations. Such an ability is especially useful for mobile applications where the resources available may change frequently, and reconfigurations are desirable for making optimal use of these resources as they come and go. Here we present a semantic checkpointing framework for use by developers to easily write adaptive applications that maintain state within a general-purpose checkpointed execution environment. A prototype implementation was built using these framework recommendations and served to demonstrate the feasibility of using semantic checkpointing as a means for enabling runtime-reconfigurable applications.

Thesis Supervisor: Karen R. Sollins
Title: Principal Research Scientist
Acknowledgments

I would like to thank:

Karen Sollins, my thesis supervisor, for all her guidance and support. She provided much valuable advice on my work, as well as on life in general. She was also a great source of encouragement, always optimistic regardless of the situation. Without her, this thesis would not be possible.

Mom and Dad, Jen and Ray. They have been there for me since day one. They are also always great to hang out with, to watch tv with, or just to eat with. Oooh yummy yummy sushi and prime rib.

Joyce, along with Chunky, Fleecie, Pat the Bunny, Paperclip, Baby-Cinnamon, and many, many, many others for their love, support, companionship, encouragement and lots and lots of coffee.

Han Sen and Alice, Zhi-Heng and Su Ying (yay I put in the dash for once!), Sean and Buddhika, Kevin and Claire, Ben and Jeremy, Woonteck and Christina, as well as any other people I may have missed out (sorry!) for their friendship. All of you have affected my life in many ways... mostly positive :). And don’t ask me why I’m randomly pairing people together. It is fun.

Yahoo and Google, for both work and play.

The Public Service Commission of Singapore, they sponsored my education here at MIT.

This work was funded in part under NSF Grant ANIR-0137403, "Regions: A new architectural capability in networking". It was also funded in part by a gift from the Cisco University Research Program.
## Contents

1 Introduction ............................................. 11
   1.1 Motivation ........................................ 12
   1.2 Semantic Checkpointing ............................. 14
   1.3 Thesis Overview .................................... 15

2 Related Work .......................................... 17
   2.1 Adaptive Software ................................. 17
   2.2 Component Software ................................. 20
   2.3 Checkpointing .................................... 21
   2.4 Summary ........................................... 22

3 Target Environment ..................................... 23
   3.1 Rationale .......................................... 23
   3.2 The Environment ................................... 25
   3.3 Pebbles and Goals ................................. 27
   3.4 Summary ........................................... 29

4 Design Considerations .................................. 31
   4.1 User Requirements ................................. 32
      4.1.1 End-user Requirements ........................ 32
      4.1.2 Developer Requirements ........................ 36
   4.2 Framework Mechanics ............................... 37
      4.2.1 Location of Checkpoint Store ................. 37
4.2.2 When to Checkpoint .................................. 40
4.2.3 Checkpointing Algorithm .......................... 41
4.2.4 Checkpointing Limitations .......................... 49
4.3 Improvements and Optimizations ..................... 57
  4.3.1 Initial Input Retention .......................... 57
  4.3.2 Timelines ......................................... 57
  4.3.3 Buffer Retention Behavior over Reconfigurations 58
4.4 Summary ............................................... 59

5 Design and Implementation .............................. 61
  5.1 Roles of Each Component ............................ 61
    5.1.1 Checkpoint Store ............................... 62
    5.1.2 Pebbles ........................................ 66
    5.1.3 Reconstituting Engine ......................... 68
    5.1.4 Planning Engine ............................... 68
  5.2 Scenarios ........................................... 69
    5.2.1 Initialization .................................. 69
    5.2.2 Regular Checkpoints ........................... 70
    5.2.3 Forced Reconfiguration ......................... 70
    5.2.4 Intentional Reconfiguration ..................... 71
  5.3 Prototype Implementation .......................... 72
  5.4 Test Application .................................... 74
  5.5 Summary ............................................. 76

6 Conclusions and Future Work ........................... 77
List of Figures

3-1 Initialization of an application in the target environment. 26
3-2 Reconfiguration of an application in the target environment. 26
3-3 A simple teleconference application in the Pebbles framework. 28

4-1 Planning engine analyses the pebble graph and instantiates one central checkpoint store for each local network 39
4-2 Timeline showing a single message being passed between two pebbles. 42
4-3 Timeline showing several messages being passed between two pebbles and some possible snapshot times relative to message times. 44
4-4 A simple application consisting of three pebbles. 45
4-5 A simple application with a cyclic pebble structure. 54

5-1 Screenshot from text source and text sink pebbles. 74
Chapter 1

Introduction

In the lifetime of a typical running application, it is often the case that the available resources will vary over time. As such, it may be desirable for applications to be modified from time to time so as to perform optimally. Even if there were no change in the resources that are available, a change in user-requirements could also necessitate the modification of an application. Such modifications could range from simply changing timeout values to adding new features. Since all such modifications are aimed at adapting an application to a new but related task, we shall refer to them here as application reconfigurations.

It is easier for applications to be reconfigured when they are built in a modular fashion, where each component is responsible for a separate portion of a program's functionality. The implementation of individual features can then be modified easily by replacing only the relevant components. Take, for example, an application attempting to draw a window on the screen. In a modern graphical operating system, the application would pass a call for the window manager (or equivalent) to draw a window. Since the application itself needs to have no knowledge of how this window should be drawn, the same application could look very different if run with a different window manager. Deeper within the operating system, we also see that the window manager itself has no idea how to interact with the display adapter. Instead this responsibility is delegated to the display driver. Similarly, changing the display driver would allow the use of a different display adapter.
It is easy to see how a modular approach to software design results in flexible programs that are easily reconfigured. However, when the components of such programs are reorganized, it is usually necessary to save all work and restart the application before old components can retire and new components can come into play. This process is time consuming and inconvenient, especially if reconfigurations are to happen frequently. Beyond this, a loss of a resource may result in the unexpected failure of a component before work can be saved. Even when the application has been reconfigured and restarted, all previous progress would have been lost.

This thesis recommends a framework that makes it possible for programs to reconstitute their components on-the-fly, and in so doing enables runtime-reconfigurable applications. The framework described here is not limited to the simple substitution of individual components that perform the same task. Instead, two completely different sets of components could replace each other as long as each set as a whole performs the same higher-level task. Under differing conditions, the same task may best be accomplished through many different combinations of sub-tasks.

The increased flexibility provided by such a framework would allow new kinds of applications to be built. For example, applications running on mobile devices could be designed to automatically and seamlessly reconfigure themselves as available resources in the vicinity change.

1.1 Motivation

In an increasingly mobile environment, the set of resources available to an application remains far from static. Simply moving from one room to the next, an application may find different types of display devices, sound devices and even network connections available to it. Without the ability to reconfigure itself on-the-fly, a program would need to restart each time new resources are to be utilized or when existing resources become unavailable. This results in an inconvenience that may outweigh any possible benefits brought about by a reconfiguration. Additionally, an unexpected loss of a resource could bring with it a loss of all progress.
On the other hand, a semantically checkpointed, runtime-reconfigurable application would be able to seamlessly adapt itself to make use of new resources when they become available. Conversely, it would also be able to transfer the last saved state to a new configuration if a key component were to fail unexpectedly.

To illustrate the usefulness of such a framework, let us consider the following scenario that could be made possible through the use of semantic checkpointing:

1. Joe User is watching a movie on his handheld. This video feed is being streamed from his home desktop.

2. He walks into his office where there is a plasma display. The application automatically recognizes this, and reconfigures itself to output to that display instead.

3. His handheld battery dies. The movie stops because there is no longer any device available to perform video processing.

4. Later, Joe turns on his laptop. The application notices this and reconfigures itself to use the laptop to stream video instead. Since the movie position was checkpointed a little before the handheld failed, the movie continues roughly where Joe left off.

This is not to say that mobile applications are the only ones that would benefit from becoming runtime-reconfigurable. As another example, a distributed computation task running on two workstations could automatically recognize the addition of another machine to the network and reconfigure itself to make use of the third machine as well. If one of these machines were to subsequently fail, semantic checkpointing would also allow the state of the computation to be rolled back to the last saved state even if no explicit backup mechanism were put in place.
1.2 Semantic Checkpointing

Checkpointing enables runtime-reconfigurable applications by keeping track of application state while components are swapped in and out. To this end, the semantic checkpointing framework described in this thesis defines how state should be captured and redeployed in the face of possibly differing sets of components. Rather than checkpointing all data values contained in each component, the proposed framework checkpoints only data that the application deems to have semantic significance across reconfigurations.

The semantic checkpointing framework presented here proposes that regular, consistent checkpoints are to be taken from all components so that there will be a state to fall back on in the event of an unexpected failure. Additionally, it describes how the application should interact with a checkpoint store to coordinate state storage and retrieval across reconfigurations.

While this framework could be implemented in applications individually, that is not the main intent of this framework. Instead, this semantic checkpointing framework becomes far more useful when used in conjunction with a general purpose planning engine that can monitor a running application and automatically reconfigure the application based on its stated tasks when it may be desirable to do so. Such a combination would allow developers to easily write reconfigurable applications. While designing a planning engine is not part of this thesis, the recommended framework was designed with such an engine in mind. In fact, the prototype implementation was integrated with one such engine.

Without such a general-purpose infrastructure, developers can already write applications that are relatively reconfigurable by predicting all the possible changes in environment that may be experienced and anticipatively coding for them. However, not only is this more difficult, it is also impossible for one to foresee everything. For example, even if one were to interface with a device using some standard, such standards may change over time and newer resources would be unusable even if they are available.
In a general-purpose infrastructure for semantic checkpointing, we envision that each application would provide a *checkpoint specification* that lists the state variables that are semantically significant to its operation. However, there may not be a simple one-to-one mapping from these state variables to the actual checkpointable variables within application components. In fact, a single state variable may be represented by data contained in multiple variables across different components. Furthermore, different configurations of an application may have different representations for the same state.

Hence, there is a need for a function that can extract and translate state from components to form some abstract application state, as well as for the inverse of this function for redistributing the collected state back to a different set of components. However, since this thesis only addresses the mechanics of performing checkpoints, we shall leave the problem of state translation to be tackled elsewhere.

### 1.3 Thesis Overview

This thesis begins with a survey of related work in Chapter 2. This includes a discussion of projects with similar goals, as well as background in component-based software and checkpointing. Earlier, it was mentioned that our checkpointing framework has in mind some general-purpose infrastructure for runtime-reconfigurable applications. Chapter 3 describes this target environment in more detail. It is also in this chapter that we introduce the system within which our prototype implementation was developed. Specific design considerations that shaped the framework are detailed in Chapter 4. Among these are the user requirements of the system and how they affect the underlying framework mechanics. The actual specification is provided in Chapter 5. This chapter additionally presents our prototype implementation together with a test application that demonstrates the feasibility of using semantic checkpointing for enabling runtime-reconfigurable applications. Finally, Chapter 6 concludes.
Chapter 2

Related Work

As computing becomes increasingly ubiquitous and users more mobile, there is a growing desire for applications to automatically adapt to changes in their environment. As a result, several research and industry efforts have been made towards making this possible. This chapter will describe some of these efforts and touch on how they relate to our own approach for enabling adaptive software.

In particular, our approach relies on the reselection and rearrangement of software components to modify the way applications behave. Additionally, our approach also relies on checkpointing for allowing reconfigurations to proceed seamlessly during runtime. Hence, related work in the areas of component software and checkpointing will also be presented here.

2.1 Adaptive Software

Dynamic environment variables [8] allow applications to be notified when changes in the environment occur. This facility is provided through the use of Dynamic Environment Servers. At startup, an application expresses its interest in various environment variables by subscribing to them on the server. Multiple applications can express interest in the same variables. When each of these variables change, the dynamic environment server notifies all interested applications, and they can choose to take action to adapt to these changes.
Sun Microsystems's Jini technology [11] provides an infrastructure for federating services together in a distributed system, as well as a programming model for how applications should be written to take advantage of such a federation. These services are pieces of Java code that represent entities such as hardware devices, computation power, data storage, or users. When a service joins a network, it uses a discovery protocol to find an appropriate lookup service to join. Similarly, applications find services on the network by sending the desired Java type to the lookup service. In addition to service discovery, Jini provides a channel for service events to be signaled to applications. Jini also provides support for transactions in distributed applications by providing an interface for a two-phase commit. However, the transactions semantics have to be implemented by the applications themselves.

Universal Plug and Play (UPnP) [2] is a system architecture contributed by Microsoft that also provides for automatic discovery of services on a network. In the UPnP architecture, services use the Simple Service Discovery Protocol (SSDP) for announcing their presence when they become available. When Control Points start up, they can also perform searches for services using SSDP. In this architecture, device and service types are identified by service description templates drawn out by individual working committees. Unlike Jini, UPnP appears to be more geared towards human users than to applications. In a sample scenario, a user would use a control point to search for a device type, and the control point would return the device URL from which the user could view device status or control the device. Device events can also be displayed to the user through control points. Conceptually, an application itself could also act as control point, discover devices, and make use of them as long as the application is familiar with their interface.

While Jini and UPnP have not yet made a significant impact, many users are already becoming familiar with the ability for wireless applications to perform the same function using different devices through the use of Bluetooth technology [10]. Bluetooth it is a networking protocol for devices to communicate within a small area. Bluetooth devices support defined service profiles [9] for each of the services they provide, and applications could use these known profiles to search for and utilize
devices in the area. With the Bluetooth devices available today, a user can already make use of the LAN access profile (LAP) on his cell phone to access the Internet. The cell phone can in turn select between its various options for connecting depending on what is available. It could even utilize the LAN access profile on yet another Bluetooth device that was beyond the range of the user's own tranceiver. Bluetooth service profiles provide an abstraction between the service provided by the device, and how this service is provided. The headset profile (HS), for example, allows one to use his desktop karaoke system as a headset for his cell phone, or even to play music on his desktop through his cell phone’s speakers.

So far, all the approaches described provide some way of notification when changes in the environment occur. Jini, UPnP and Bluetooth additionally provide applications with a way to select and utilize different devices for the same purpose as long as they provide the same Java interface, service description template, or service profile. However, the mechanics of such a switch is left either to the application developer to implement, or to the user to quit the application and restart it with a different selection of services. Moreover, these systems tend to advocate a simple substitution of devices or services, rather than an structurally different combination of services as is the case in our solution.

A system that is more in line with our objectives is the Goals [14] system developed here at LCS. The Goals systems provides a way for applications to be written as sets of goals. At startup time, the Goals resolution engine analyzes these goals, evaluates the resources available, then constructs a suitable implementation of the application. Since these goals are only interpreted when the application is started, this allows the application to be optimized for each instantiation. However, even as resources in the environment change, the application cannot adapt to these changes until its next instantiation.
2.2 Component Software

As mentioned earlier, our approach to enabling reconfigurable applications is through the reselection and rearrangement of components that make up the application. This leads to an interest in systems that allow one to build distributed applications by connecting simple components together.

The Programmers’ Playground [3, 12] is a system that allows end-users to create distributed multimedia applications by connecting modules that were written using the I/O abstraction model. In this model, modules may contain a set of variables declared using a special set of Playground data types. Such variables can be published, and communication is established by defining connectors between the variables of different modules. The Programmers’ Playground also uses these publishable variables to enable semantic migration of modules.

Ajanta [13] and Hive [7] are some examples of systems that allow the building of mobile agent-based applications. Both of these specify mechanisms for the activating and deactivating of agents, messaging between agents as well as migration of agents between hosts. While Ajanta utilizes object serialization for maintaining state as the agents move, agents in Hive must make their own arrangements for the transfer of state.

The Pebbles [15] software component system is of particular interest because it was developed with the Goals system in mind. In the Pebbles system, applications are built through the assembly of very basic software components, or pebbles. Each of these pebbles only performs a limited set of tasks. However, connecting these pebbles together allows for the construction of complex applications. When combined with the Goals system, each instantiation of an application is created by selecting a suitable set of pebbles and joining them together.

After a review of these software component systems, the Pebbles system was chosen as the model environment for our semantic checkpointing framework. The reason is that its functionality when combined with the Goals system seemed more in line with our objectives. Specifically, the reevaluation of goals after a change in
the environment would result in a different configuration of pebbles that perform the same higher-level function. Additionally, the availability of the source code for the Pebbles environment allowed us to extend it and built a prototype checkpointed environment for runtime-reconfigurable applications.

It should be noted that all the other software component systems mentioned here provide mechanisms for process migration. Process migration has long been recognized as one way of adapting applications to changing resources [4]. For example, processes in a distributed system could be migrated to lightly used machines for load balancing. However, the flexibility of adaptations allowed by process migration is limited compared with the possibilities presented by the Pebbles and Goals environment.

2.3 Checkpointing

Checkpointing forms the basis of how our framework enables state persistence across application reconfigurations. Traditionally, checkpointing has been utilized to provide fault tolerance in systems. Of special interest to us are checkpointing algorithms for distributed systems.

It is accepted that for a set of checkpoints to be consistent and usable, all snapshots must be concurrent with each other. The widely accepted view of concurrency is based on Lamport’s ‘happened before’ relation [6] which provides an ordering of events in a distributed system. If event \( a \) occurs before event \( b \) in the same process, then \( a \) happened before \( b \). Also, if \( a \) is the sending of a message in one process, and \( b \) is the receipt of that message in another process, then \( a \) happened before \( b \). Hence, two events, \( a \) and \( b \) are concurrent if neither \( a \) nor \( b \) happened before the other.

Chandy and Lamport’s distributed snapshot algorithm [1] further forms the basis of many other checkpointing approaches. This algorithm makes use of message-passing between processes to coordinate the timing of snapshots among processes so that all snapshots taken are concurrent with each other. A comprehensive survey of checkpointing algorithms available is given in [5].
We draw upon these accepted checkpointing algorithms in the design of our framework. However there is one fundamental difference in our requirements that should be pointed out here: while traditional checkpoints are meant to capture and restore state to the same sets of components, our checkpointing algorithm will need to handle changes in the entire application structure.

2.4 Summary

Here we highlighted several related projects that also aim to enable automatic adaptation in applications. However, while these works have similar objectives as our framework, they take approaches that differ from ours of checkpointing component-based software. This is not to say that the two concepts of component-based software and of checkpointing are new. On the contrary, much work has been done in these areas and we build upon this in the design of our semantic checkpointing framework.
Chapter 3

Target Environment

Earlier, it was mentioned that while semantic checkpointing could be implemented into applications individually, this is not main intent of the framework presented here. Instead, this semantic checkpointing framework is meant to be used within some generic execution environment where a common engine would be responsible for automatically monitoring and reconfiguring applications.

This chapter provides a discussion of the target environment; including the rationale behind choosing such an environment, scenarios of how such an environment would work, and a specific example of one such environment.

3.1 Rationale

Compared with regular applications, runtime-reconfigurable applications have three additional duties: planning when and how components should be rearranged, performing the actual rearrangement of components, and checkpointing of state across rearrangements. These actions shall be referred to here as planning, reconstituting, and checkpointing respectively. Since all runtime-reconfigurable applications would need to perform these actions, it would be desirable to have an execution environment that carried these duties out for them instead.

There are several distinct advantages to having an execution environment for runtime-reconfigurable applications and some of these are given below:
• First of all, since key functions are already taken care of by the environment, it would be easier for runtime-reconfigurable applications to be written for such an environment. This, in turn, would promote the proliferation of runtime-reconfigurable applications.

• Runtime-reconfigurable applications need to constantly monitor the resources available to decide whether a reconfiguration would be desirable. If planning engines were to be implemented into applications individually, this would mean that there would be one planning engine for each application and that all of them would be performing the same task of monitoring the available resources. On the other hand, the usage of a common execution environment would allow all of this monitoring to be performed by a single planning engine hence freeing up resources for other tasks.

• Usage of a common execution environment also allows applications to evolve with time. After an application has been released, new kinds of components with different interfaces may be developed allowing for new combinations of components to perform the same task. While an application developer cannot anticipatively code for such future components, he could instead rely on the common execution environment being constantly upgraded to take advantage of new types of resources.

• When numerous applications rely on a common execution environment for runtime-reconfigurability, the large usage of this common execution environment provides more opportunities for review and results in a stronger impetus for the environment to be implemented correctly and efficiently. As a result, applications written for a common execution environment are also likely to reconfigure themselves more reliably.

While this list of advantages mentioned is not exhaustive, it is enough to suggest that runtime-reconfigurable applications would benefit from having a common execution environment that serviced them in terms of planning, reconstituting, and
checkpointing. Enabling runtime-reconfigurable applications is the key purpose of the framework presented here and hence it makes sense for the framework to be designed with such a target environment in mind.

3.2 The Environment

In the target environment described, the tasks of planning, reconstituting, and checkpointing must somehow be abstracted away from the rest of the application while still allowing the application to function normally.

Of these tasks, checkpointing and reconstituting are fairly generic procedures that do not need to vary greatly from application to application as long as the components used have a common interface for key operations such as being started and stopped, for reporting and accepting state, and for redirecting its various inputs and outputs.

On the other hand, the job of planning is more complicated since each application has a different set of functions it needs to perform, and knowledge of these functions is essential for deciding what other arrangements of components may be suitable. Hence, in order for a generic planning engine to be able to carry out planning on behalf of an application, the application must first specify what tasks it needs performed. Now, since the planning engine has taken over the responsibility of coordinating components, the application itself no longer needs to have any awareness of the components it is using. Where the application once used to interact with multiple components, this is now replaced by an interface with a single planning engine. Even as planning, reconstituting, and checkpointing engines interact with components, this is handled by the planning engine and thus appears completely transparent to the application.

In this way, when an application is first instantiated, it sends the set of tasks that it needs performed to the execution environment. The planning engine in the execution environment receives this set of tasks and decides on a suitable configuration of components. This configuration is relayed to the reconstituting engine which then causes this configuration of components to come about. This process is shown in Figure 3-1. Now, the planning engine sets about monitoring the application, components, and
1. sends a list of external tasks

Figure 3-1: Initialization of an application in the target environment.

2. event is detected
1. some change in available resources occurs

Figure 3-2: Reconfiguration of an application in the target environment.
the available resources. If a condition is detected that warrants a reconfiguration, the planning engine sends the new configuration to the reconstituting engine, which in turn proceeds to rearrange the components. This is shown in Figure 3-2.

3.3 Pebbles and Goals

In the process of designing the semantic checkpointing framework, it was useful to have a specific execution environment in mind in order to keep the design practical. For this purpose, we chose to build on two projects undertaken by the Computer Architecture Group here at LCS: Pebbles, and Goals [14, 15].

The Goals project proposes goals as a formal language construct, with goals replacing regular procedure calls. While procedures can be thought of as little goals themselves, the key difference between procedures and the proposed goals construct is that while procedure definitions are accompanied by a block of code, goals are completely disembodied from their implementation. Instead, a goal resolution engine seeks to satisfy each goal by searching for a particular set of techniques that will satisfy the goal. Each of these techniques could in turn comprise of sub-goals and these sub-goals are recursively resolved until the first goal has been fully satisfied by a set of techniques.

Techniques themselves are similar to procedures in that they comprise a specific implementation. However, techniques do not specify any specific goal. Instead, they specify a pattern that loosely describes their function and this pattern is matched against candidate goals.

This goals language construct and resolution engine could play the role of the planning engine in our target environment where techniques are viewed of as components. Indeed, the application itself could simply comprise of a single goal. An example of this is given in [14] where an application could consist of a single high-level goal such as \texttt{Teleconference(Victor, Steve)} and the resolution engine would set about finding techniques to make this possible.

The Pebbles project completes the picture by providing a system that allows dis-
Pebbles are basic components that perform a single specific function and may optionally contain several input and output channels. It is convenient to view pebble-based applications in the form of a graph, with pebbles forming the nodes and their input and output channels forming the edges.

Since the Pebbles System works across a distributed network, the teleconference application could be written by simply installing an audio source and an audio sink pebble in each of the two machines as shown in Figure 3-3. This application could easily be turned into a voice-activated portable music player for Victor by replacing the audio source pebble on Steve’s machine with a file streaming pebble and replacing the audio sink pebble with a voice-recognition pebble.

In a sense, each of these pebbles could be viewed of as a technique in the Goals system. If the Pebbles and Goals systems were combined, distributed applications could be written as a set of high-level goals with the resolution engine breaking down goals into a combination of techniques, or pebbles; and the Pebbles System inserting these pebbles into machines across the network, connecting them together, and in so doing causing the distributed application to come into existence.

However, while the combination of the Pebbles and Goals systems would allow the creation of applications on-the-fly, this alone would not allow applications to be reconstituted without restarting. This is where the proposed checkpointing frame-
work enters the picture. Together, the planning and reconstituting engine would adopt the recommendations of the proposed framework to form a complete execution environment for runtime-reconfigurable applications.

In the rest of this paper, we will borrow the term 'pebble' when referring to the basic components that form an application in our target environment. The usage of this term serves to differentiate application components from the other components in the execution environment, and is not meant to tie our framework specifically to the Pebbles System. Additionally, it highlights the simplicity of the components we have in mind.

### 3.4 Summary

Our goal for designing a semantic checkpointing framework was to enable runtime-reconfigurable applications, and choosing an appropriate execution environment is a key step in achieving this goal.

The target environment described helps make it easier to build efficient and reliable runtime-reconfigurable applications that can evolve over time even as new devices are developed along with new methods for doing things.

In particular, the Pebbles software component system and the Goals resolution engine were selected to work together with our proposed checkpointing framework. It is intended that this combination will form a complete execution environment for runtime-reconfigurable applications.
Chapter 4

Design Considerations

In order for semantic checkpointing to be possible in any execution environment, it is necessary for all components in the environment to follow a predetermined checkpointing protocol, where this protocol defines the role of each component in ensuring that global state can be captured and redeployed as necessary.

In the context of our target environment, this protocol should define the duties of and interactions between the application script, planning and reconstituting engine, program components, as well as any additional component that may be introduced to enable persistent state. It is these duties and interactions that form our semantic checkpointing framework.

There are many different ways of laying out these duties and interactions while still achieving the same end-goal of enabling persistence of state over application reconfigurations. However, each of these approaches would produce various different mixes of performance, usability, flexibility and simplicity in the resultant system.

With this in mind, this chapter discusses some of the more important design considerations that helped shape the final framework recommendation before proceeding to see how everything comes together with the design specification in the next chapter. Of course, one could also proceed in the opposite order, reading the next chapter first to gain an in-depth view of the system mechanics before reading this chapter to understand the rationale behind it all.

This chapter will first describe our target audience in order to understand some
of the specific goals of the system, then it will show how these goals shaped the basic design of the framework. Finally, it will end with a discussion of some possible improvements and optimizations to the basic design.

As was first mentioned in Section 3.3, we will continue to use the term pebble to refer to the basic components that combine to form an application. This usage is not meant to tie our recommended framework specifically to the Pebbles System. Instead, it highlights the simplicity of the components we have in mind. Also, this usage serves to differentiate between these basic building blocks, and other components that will participate in the framework.

4.1 User Requirements

Two main groups of people would be affected by our semantic checkpointing framework: end-users and developers.

4.1.1 End-user Requirements

For end users, the benefit of having a semantic checkpointing framework would be the availability of applications that can automatically and seamlessly reconfigure themselves as available resources change. While the average user may not have any knowledge of the inner-workings of these applications, the actions performed by the underlying checkpointing framework could still surface in ways that would affect the end-user. Such effects would be apparent during reconfigurations, as well as during normal operation.

Forced Reconfigurations

The planning engine may initiate a forced reconfiguration when resources in use unexpectedly become unavailable and some pebbles fail as a result. For example, if a hardware component or network path between two elements becomes unavailable, some alternative configuration will need to be arranged. Forced reconfigurations may also be the result of changes in security restrictions.
During a forced reconfiguration, the checkpointing framework would not have the opportunity to capture the most recent state from failed pebbles and hence the resultant reconfiguration of an application would have no choice but to revert to the last saved checkpoint.

In such a situation, the alternative without semantic checkpointing would be to restart the application anew and repeat all lost work. Hence, for semantic checkpointing to be useful in this case, the benefit in terms of time and progress saved would have to be greater than the cost of starting over. For starters, this means that the checkpoint information saved would have to be correct and consistent. Suppose this were not the case and the application was reconfigured with an invalid state, the user would at best have to discard the saved state and start over. At worst, the user may not even notice the invalid state and would continue to work in an environment that is producing incorrect results.

The usefulness of the saved state may also depend on how recent it is. If the state is too old, an application may be better off starting anew. Ideally, the saved state would be the exact state of the system before the pebble failure occurred. However, the cost associated with continuously logging current state may result in poor performance to the extent of making the application unusable.

Related to this is the cost associated with redeploying state into a new configuration of an application. Indeed, a user may prefer his chat application to quickly restart anew and let him continue his conversation rather than spend time trying to reconstruct a log of all past exchanges. At the same time, other applications a user may run may have state that changes frequently only during initialization and for such applications, any post-initialization snapshot would be more useful than having none at all. An example of such an application would be the case of GPS receivers where the initial location fix could take minutes, while future location updates could take seconds as long as some state from a previous location attempt is known.

Here we see three further user requirements: the need for fresh state, a minimal performance hit associated with checkpointing, and the quick restoration of state to new configurations of applications. However, these goals are closely linked and achiev-
ing the first often means sacrificing the latter two. Since the balance of importance between these goals varies with the application being run by the user, the user may further desire that applications be able to tune the importance of these goals in the framework according to its needs.

Finally, the user may also have certain expectations of how his software should resume after a forced switchover. The time that the forced switchover takes represents an interruption, and during this interruption certain real-time events may have taken place. During this interruption, it may be possible for the application to buffer all events to be replayed after the interruption. However, the user may or may not find it desirable for such replay to take place depending on the application he is running, or the way in which he is using the application. For example, a user streaming a movie to his handheld from his home computer may want the movie to continue where it was just before the interruption. On the other hand, if he were streaming video off a TV news channel, he may prefer that the application to jump straight to the most recent footage instead.

**Intentional Reconfigurations**

Aside from initiating a reconfiguration if pebbles unexpectedly fail, the planning engine may also initiate a reconfiguration if a possibly 'better' configuration becomes available. A full discussion of how exactly a planning engine may decide which configurations are 'better' than others is outside the scope of this thesis. Briefly, some situations whereby a planning engine may initiate an intentional reconfiguration would be if parts of the current configuration appears to be degrading in performance and on the verge of failure or if new resources become available and the planning engine decides to make use of them.

One alternative without semantic checkpointing would be for the application to simply continue without any changes. Since the old configuration is still functional when an intentional reconfiguration occurs, this is actually a feasible option. After all, users are already accustomed to such behavior. The other alternative would be for the user to save his work before manually performing a reconfiguration of his software.
However, there are distinct benefits to allowing an application to automatically reconfigure itself for the better and if the added value of the new configuration is greater than the switchover cost, then this reconfiguration would have been worth the effort for the end-user. Since this thesis will not touch on how this cost-benefit analysis is done, minimizing the switchover cost associated with application reconfigurations will be the focus of this section.

Most importantly, an intentional reconfiguration should not result in any loss of data or progress in the application. After all, this is already guaranteed by both the alternatives mentioned above.

Additionally, the reconfiguration should also cause minimal interruption in the user’s work. One can envision a mobile user frequently moving through many different environments each having its own optimal configuration. If reconfigurations caused a significant interruption, this would make the application unusable.

While intentional reconfigurations would ideally happen seamlessly, this may not always be possible. Instead, there may be a lag between the instant when old pebbles leave the program structure and the time when new pebbles are activated as part of the new program structure. This interruption has similar consequences to that discussed earlier for the case of forced reconfigurations. However, the two different types of reconfigurations may require different measures for ensuring that the users’ desired interruption behavior is achieved.

Furthermore, there may also be some differences in a user’s desired interruption behavior for the two types of reconfigurations. For example, in the case of forced reconfigurations, pebbles may fail long before an alternative reconfiguration becomes available. This means that an intentional reconfiguration interruption could be much shorter than that of a forced reconfiguration. With this shorter interruption, the user streaming a movie to his PDA may now prefer that the application skip the few frames that would have been played during the interruption instead since a time-shift in video playback would contribute to a perception of jerkiness.
Normal operation

During normal operation, the user does not immediately gain from semantic checkpointing even though this is the time when checkpoints are being taken for any possible forced reconfiguration that may happen later. At the same time, the action of checkpointing is bound to have some impact on each application's speed and performance.

As such, the negative impact of semantic checkpointing on normal operation needs to be balanced with the necessity of having recent checkpoints. Since these two requirements will vary greatly with each user and each application, the semantic checkpointing framework would need to allow users to tune how frequently checkpoints should be taken.

Aside from slowing down the general operation of an application, the action of checkpointing may also make the execution of the application appear jerky. The reason is that the performance hit of checkpointing will vary with time. During a checkpoint, the application may slow down but after each checkpoint, the application would speed up to its regular pace. This jerkiness of execution speed may be tolerable for some applications but may not be so for others. Hence, the semantic checkpointing framework should specify a checkpointing mechanism that either eliminates jerkiness, or makes it user-configurable such that jerkiness could be minimized if the application requires it.

4.1.2 Developer Requirements

Several groups of developers would be involved directly and indirectly in the making of each runtime-reconfigurable application. These include developers of the execution environment, the pebbles, the application script, as well as any external checkpointing component. While not all of these developers may be directly involved with implementing the mechanics of semantic checkpointing, all of them will need to be aware of the effects of the underlying checkpointing framework.

With so many people involved, it is important that the semantic checkpointing
framework be easy to understand and that the roles of each group of developers be clearly defined. If this were not the case, it would be easy for each developer to interpret the framework differently and come up with components that are not compatible with those from other developers. Even if they appear compatible, a complicated checkpointing framework could make it easy for an implementation error in a single component to bring down the entire system.

Beyond this lies the question of how tasks should be divided among the various components. Here we argue that as much of the implementation effort as possible should lie with the components of the execution environment: the planning engine, the reconstituting engine, as well as any additional checkpoint engine. After all, it is these components that will be reused the most across various applications. Also, it is likely that these three components will be provided by the same execution environment vendor. In choosing an execution environment, pebble developers would cater to an environment that is popular among application developers while application developers themselves would want an environment with a large selection of available pebbles. At the same time, both application and pebble developers would be looking for ease of implementation among other things when choosing an execution environment. Hence, it is in the execution environment vendor’s best interests to choose a checkpointing framework that requires little from each pebble and each application.

4.2 Framework Mechanics

In order to design the semantic checkpointing framework, we identify three key issues to address: management and storage of the checkpointed data, the timing of checkpoints, and the checkpointing algorithm. After we discuss these, we will consider some limitations of the chosen algorithm.

4.2.1 Location of Checkpoint Store

In deciding where state should be stored within a semantic checkpointing framework, we are faced with several options. State could be stored together with the planning
engine, distributed among the pebbles, or in some separate entity altogether. Where-
ever the state is stored, some mechanism must exist for copying state from running pebbles during checkpoints and for redeploying state to pebbles after reconfigurations.

Since the checkpoint store must be able to communicate with all the pebbles, it would be reasonable to make the planning engine also serve as the checkpoint store. After all, the planning engine already needs to contact pebbles to initialize the application, as well as to monitor the application and decide whether and when reconfigurations should occur.

One may argue against centralized storage of state since the failure of the single checkpoint store could make a reconfiguration impossible. After all, regular checkpoints are used only in the event of a failure. A more elaborate mechanism could be suggested that distributes state throughout the pebbles in such a way that even if a subset of pebbles were to fail, state could be reconstructed from the remaining pebbles. However, the targeted execution environment already has an existing central point of failure in the planning engine and if this engine were to fail, there would be no entity to initiate and perform reconfigurations anyway. Also, if a distributed checkpoint store only managed to reconstruct partial state, this would not be useful as the resulting checkpoint would not be consistent with itself. Finally, a distributed store would also increase the complexity of pebbles since they would have to organize snapshots among themselves and work together to reconstruct global state in the event of a failure. This increased complexity would make implementation difficult for pebble developers and hence would not be desirable.

Another argument against a centralized checkpoint store is that not all pebbles may have a fast and reliable network connection to the central checkpoint store. For example, an application could be configured in such a way that its pebbles are distributed across two networks that have a slow inter-network link. In such a configuration, the planning engine may not have any problem since its interaction with the pebbles does not involve the transfer of much data. On the other hand, the checkpoint store will regularly receive potentially large checkpoints from each pebble. While snapshots from pebbles within the same network will arrive quickly, snapshots
Figure 4-1: Planning engine analyses the pebble graph and instantiates one central checkpoint store for each local network.

Arriving from pebbles in the other network would be held up due to the slow link speed the two networks. This low-speed link will limit how recent a checkpoint can be due to the time taken for snapshots to travel between the networks. Furthermore, if this inter-network link becomes congested with the high rate of snapshot transfer, messages to and from the planning engine may not be delivered reliably. Pebble connectors that span the two networks would also be affected and the entire application could slow down as a result.

One approach to this problem is to abandon the idea of a centralized checkpoint store in favor of distributing checkpoints among the pebbles. However, this is not desirable due to the reasons already stated above. Another alternative is to distribute the checkpoints for some combination of efficiency and reliability, but at locations separate from the pebbles. From our perspective at present, this is no different from a centralized service – it is simply a remote service.

To get the best of both, it was decided that the planning engine should recognize
such network complications and deploy one checkpoint store in each network as shown in Figure 4-1. The checkpoint stores should thus be able to communicate among themselves to synchronize checkpoints as well as reconstruct application state when necessary. Furthermore, the hosts chosen for each checkpoint store should ideally be chosen to be at least as reliable as the planning engine itself. Within each region, this could mean having a few dedicated checkpoint stores.

4.2.2 When to Checkpoint

Determining when checkpoints should take place is an important step in shaping the checkpointing framework. Ideally, a checkpoint should be taken just before each reconfiguration. However, if the reconfiguration is unexpected then such a checkpoint is impossible and the last checkpointed state will have to be used.

Thus an application needs to be checkpointed regularly during normal operation in order for there to be a state to fall back on in the event of a forced reconfiguration. If these checkpoints are infrequent, the recovered state may be too stale to be usable. At the same time, frequent checkpointing may result in a large performance hit.

Furthermore, different applications will have varying optimal rates of checkpointing. Critical applications may require a checkpoint after each and every transaction. At the same time, applications with spaced out bursts of activity may want checkpoints to take place only in between each of these bursts. Yet others may require checkpoints to happen at fixed points in time so that the recovered state is never older than that period.

Since these requirements vary greatly, what we propose is that the planning engine should designate a pebble to initiate checkpoints. This gives the planning engine the flexibility to decide when and how often checkpoints should be taken relative to the progress of the application. If checkpoints are required after each transaction, a pebble involved in the transaction can initiate a checkpoint after each step. If less frequent checkpoints are required, the pebble can initiate a checkpoint after every few transactions. Similarly, the planning engine can also instruct a pebble to initiate checkpoints only after a certain period of inactivity.
4.2.3 Checkpointing Algorithm

During normal operation, an application will periodically take *regular checkpoints* that are used in the event of a forced reconfiguration. Aside from regular checkpoints, *special checkpoints* may also be taken at the initial stage of an intentional reconfiguration so as to allow transfer of state from leaving pebbles to joining pebbles. Both of these types of checkpoints require an algorithm that determines how the planning engine, checkpoint store and application pebbles should work together to construct a complete application snapshot. Such an algorithm will have to produce correct checkpoints, have a minimal impact on performance as well as be easy for pebbles to implement. Among these three requirements, checkpoint correctness is the most important requirement since an invalid checkpoint would be useless regardless of how fast or easy it was to obtain.

**Checkpoint Correctness**

Each global checkpoint is an agglomeration of state from each of the pebbles, and a correct checkpoint is one composed of pebble state snapshots that are consistent with each other. As an application proceeds, the state of each pebble is constantly changing and is affected by its previous state, internal operations, as well as any inputs into the pebble. Each change of state may also be accompanied by an output which will trigger another change of state in the receiving pebble, and this chain of events continues as messages get passed through the application.

In order to illustrate how an inconsistent checkpoint could come about, let us consider a simple application instantiation with two pebbles, A and B, one connector going from B to A, and a single message being sent from B to A. Considering the timeline as shown in Figure 4-2, if the snapshot from B is taken at time $t_{b1}$ before the message is sent and the snapshot from A is taken at time $t_{a2}$ after this message is received, the resulting checkpoint would be inconsistent because it would reflect that A had received a message that was never sent by B. Such a message is known as an *orphaned message*. 
In the example given, the resulting checkpoint was not consistent because the snapshots from each pebble occurred at different times with respect to the message from B to A. For a global checkpoint to be consistent, it must be composed of pebble snapshots that were taken concurrently. The widely accepted view of concurrency is based on the "happened before" relation, denoted by $\rightarrow$, as defined by Lamport [6]:

1. If $a$ and $b$ are events in the same process and $a$ comes before $b$, then $a \rightarrow b$.

2. If $a$ is the sending of a message by one process and $b$ is the receipt of the same message by another process, then $a \rightarrow b$.

Using this, one can determine the concurrency of individual pebble snapshots by inspecting their relation in time to all messages passed in the system. A global checkpoint is consistent if all pairs of pebble snapshots are concurrent. For each pair of pebble snapshots $P_i$ and $P_j$, this is satisfied if $P_i \rightarrow P_j$ and $P_j \rightarrow P_i$.

Revisiting Figure 4-2 with this definition, we find that a snapshot from A taken at time $t_{a1}$ is concurrent with one taken from B at time $t_{b2}$ even though they were taken at different times relative to the message. The time $t_{a1}$ and $t_{b2}$ could very well be exactly the same, but since messages take time to travel, a message sent just before $t_{b2}$ could still be received after $t_{a1}$ and B would have logged the sending of a message that was not received by A. Such a message is termed a missing message.

Traditional checkpointing approaches deal with this by logging missing messages as part of the checkpoint. After recovery, these in-transit messages can then be resent.
However, such checkpoints are not useful in our checkpointing framework, because a reconfiguration may result in a new program structure that involves a very different set of pebbles with different individual functions even though they combine to achieve the same higher-level goals. In the new structure, messages that are passed between pebbles may have no relation to those passed in the old structure and hence any messages that were logged as part of a checkpoint are likely to become meaningless after reconfiguration.

**Independent Snapshots**

One approach to creating global checkpoints is for each pebble in the application to take independent snapshots periodically. When a forced reconfiguration takes places, a set of these snapshots will be selected and pieced together to form a consistent global state and this will be the state that is restored to the new configuration. This approach is attractive because it requires no coordination among pebbles during normal operation. Furthermore, since each pebble can take checkpoints at its own convenience, performance of the application is affected minimally. Finally, the simplicity of this checkpointing method makes it easy for pebbles to implement.

One drawback of this approach comes from the fact that the most recent snapshot from each pebble may not necessarily contribute to a consistent global state. Instead, one would have to iteratively rollback the selected snapshot from each pebble until a complete set of concurrent snapshots is found. This may lead to a cascading rollback and the most recent consistent checkpoint that can be assembled may be too old to be useful. This situation is known as the domino effect [5]. Since it is not immediately clear which pebble snapshots will be useful, the checkpoint store needs to save all of them initially, leading to an increased logging overhead. Finally, while there exist many methods for piecing together a consistent global checkpoint from individual independent snapshots at recovery time [5], these are fairly complex and result in an increased recovery time.

In an application where computations occur in bursts and there are long time windows when no messages are being passed, it is likely that very few pebble snapshots
Figure 4-3: Timeline showing several messages being passed between two pebbles and some possible snapshot times relative to message times.

will be useless and the recovery algorithm will find a consistent checkpoint quickly. On the other hand, if messages are being passed frequently between pebbles in an application and pebble snapshots are not being taken often enough, it may well be the case that a consistent global state cannot be found from the set of snapshots. Such a case is shown in Figure 4-3.

If applications in the environment are expected to execute occasional bursts, then this checkpointing algorithm may be suitable because of its simplicity. After all, even though the recovery cost may be high, this is incurred only during a failure. However, since this checkpointing framework is being proposed for a general execution environment, one must expect a whole range of applications and a checkpointing mechanism that fails for certain applications is not acceptable.

Synchronized Snapshots

An alternative approach is to have all the pebbles take checkpoints at approximately the same time [5], either by having synchronized clocks or by having the checkpoint instruction issued by a third party (effectively acting as a single clock). In order to account for differences in time, messages could be delayed during the checkpointing window.

However, because of our special requirement that messages cannot be logged as part of a checkpoint, there is no guarantee that a consistent checkpoint can be taken
at any instant. To illustrate this point, let us consider three pebbles $A$, $B$ and $C$ strung together as shown in Figure 4-4. Each of these pebbles receives a number from their input and outputs that number plus the previous number received. $C$ receives numbers from $B$, $B$ from $A$, and $A$ receives input from some external source. Now, suppose a checkpoint were to be attempted at an instant when $B$ and $C$ are idle, while $A$ has just performed an addition and has placed the result in its output queue. In systems where checkpoints can include messages, the snapshot of $A$ could include the outgoing message to $B$ while the snapshots of $B$ and $C$ will merely contain their state. Upon recovery, the logged message would then be resent to $B$. In our case, since checkpoints cannot contain messages, $A$ would have to send the message to $B$ before a snapshot can be taken. Now that $B$ has received a message and is processing it, it does not reach a stable state until an output has been generated. When this output is generated, $B$ finds itself in the same situation that $A$ was in initially and then the same cycle continues with $C$. Hence, $A$, $B$ and $C$ cannot be checkpointed together until $C$ has sent off its output to the external sink. If inputs to $A$ always arrive just before $C$ manages to send off its output, then the application will never reach a suitable state for synchronized checkpointing.

The approach to this problem would be to let each pebble flush its output as well as to stall all external inputs whenever a checkpoint needs to be taken. This ensures that a stable application state absent of in-transit messages can be achieved. If any pebbles in the system are data sources themselves, these should also be paused so that new inputs into the system are not generated.
An advantage of this method of checkpointing is that it produces correct results and is simple to implement. However, the stall necessary to achieve a stable state before each checkpoint can be taken would have a negative impact on performance. If employed in regular checkpointing, this checkpointing algorithm would result in an application with frequent and noticeable stalls. This would be especially pronounced in applications with long pipelines of pebbles as the stall duration would be equivalent to the sum of execution time for all of the pebbles in the pipeline.

On the other hand, this approach is attractive when checkpointing for an intentional reconfiguration. First of all, such checkpoints will not occur as frequently as regular checkpoints. Secondly, the leaving set of pebbles would need to have their output flushed and any in-progress operations completed anyway before control can be passed over to the joining set of pebbles. Since not all the pebbles need to be flushed during an intentional reconfiguration, this checkpoint could be limited to just the leaving set of pebbles, and any inputs from other pebbles could be treated as external sources.

Coordinated Snapshots

Rather than trying to synchronize checkpoints according to one clock, it may be beneficial to take an alternate definition of time, and view time as relative to the sequence of messages input into the application instead. Each incoming message can be regarded as a point in time, and as these messages are transformed and propagated down the application, they still represent the same time. Individual pebbles can then decide when to take checkpoints based on when messages are received rather than based on a synchronized clock. In this way, snapshots from pebbles can be coordinated based on the messages passed through the application.

In the same example with three pebbles $A$, $B$ and $C$, one can imagine each message sent to $A$ carrying a certain time marker. The pebbles could agree to take snapshots immediately after processing the $i$th message, and when these snapshots are pieced together, the resultant checkpoint would then represent the steady state that the application would have achieved if the $i + 1$th message were never sent.
It turns out that this is similar to what is done in Chandy and Lamport’s distributed snapshot algorithm [1]. In this algorithm, markers are sent along channels to coordinate checkpointing. Upon the first receipt of such a marker, each participant in the algorithm would take a local snapshot of its state. This snapshot is guaranteed to be concurrent with that taken from the sender of the marker if the channel between the two is FIFO. The reason for this is that the FIFO nature of the channel guarantees that the messages sent before the marker, pre-marker messages, are received before the marker, and that messages sent after the marker, post-marker messages, are received after the marker. Since each node’s snapshot is brought about by the receipt of the marker, and each node’s snapshot is concurrent with the marker sender’s snapshot, by induction this guarantees that all snapshots, taken in a network where the indegree of each node is at most 1, will be concurrent with that of the snapshot that generated the first marker. If we assign the first snapshot a timestamp, this means that all snapshots taken will have the same timestamp.

The question of when to take snapshots is slightly more complicated when a node has an indegree of more than one, because the node will receive a checkpoint marker from each of its input channels at different times. Chandy and Lamport’s approach to this is for the node to take its snapshot upon the first receipt of the marker on any channel. For each of the other channels, all incoming messages will be logged until the marker is received. This is akin to the logging of missing messages for replay after recovery.

In this way, Chandy and Lamport’s distributed snapshot algorithm defines how each node should behave upon first receiving a marker. The algorithm is given as follows:

1 save state to snapshot upon receiving marker \( m \)

2 propagate \( m \) to all outgoing channels

3 while continuing regular computation, log all incoming messages from each channel \( c \) to the snapshot until the marker \( m \) has been received on \( c \)
In step 3 of the algorithm, we see pre-marker messages arriving on other channels being logged so that they can be replayed after recovery. However, this cannot be done in our framework due to the special requirement mentioned earlier that messages cannot form part of a checkpoint. At the same time, this step is necessary to ensure that all pre-marker messages, and no post-marker messages are incorporated into the checkpointed state. There are ways around this and these will be discussed in the next section.

An interesting feature about the resultant checkpoint is that the saved state is as though none of the messages after the initial marker entered the system. This is similar to the result with synchronized checkpointing as described above when inputs are stalled for the duration of the checkpoint, yet no inputs are stalled in this case. This absence of stalls results in minimal performance impact even though each checkpoint itself still takes as long as the pipeline execution time.

Chandy and Lamport’s distributed snapshot algorithm has several distinct advantages. First of all, it allows computation to continue while checkpoints are being taken. Secondly, every snapshot taken is guaranteed to be part of a consistent checkpoint so no time and bandwidth is wasted in taking extraneous snapshots. Finally, the algorithm is simple for each pebble to implement yet piecing the snapshots together to form a consistent state is still simple for the checkpoint store. Assuming that an alternative for message logging can be found, this checkpointing approach is definitely desirable for regular checkpointing in our framework. Whenever a checkpoint is to be taken, the checkpoint store could generate a timestamp and instruct all root pebbles in the program structure to propagate checkpoint markers bearing this generated timestamp. As each pebble’s snapshot is completed, this could be sent to the checkpoint store together with its associated timestamp.

Summary

So far, we have settled on using a variant of synchronized checkpointing for special checkpoints while Chandy and Lamport’s coordinated checkpointing approach seems
best suited to our needs for regular checkpoints. Since messages cannot be logged as part of the checkpoints in our framework, this will be left out of the regular checkpoint algorithm for now.

4.2.4 Checkpointing Limitations

The checkpointing algorithm presented so far is acceptable for simple arrangements of pebbles. However, this algorithm has its limitations and there are certain situations that will result in unusable checkpoints. Such arrangements and some workarounds will be discussed here.

Pebbles with Multiple Inputs

The message logging portion of the Chandy and Lamport algorithm is triggered only when multiple inputs are present. Expectedly, disabling message logging in our algorithm thus produces unusable checkpoints when pebbles are instantiated with more than one input.

An approach to this problem is to recognize that with the original algorithm, any logged messages would be sent to the recipient after recovery and in so doing affect the recipient’s state. Rather than waiting till after recovery to allow this to happen, one could take a snapshot of the recipient pebble only after it has received all the messages that would have otherwise been logged. In this way, logged messages are not logged in their raw form, but are instead incorporated into the checkpointed state. This is equivalent to taking the snapshot after all markers have been received. Unfortunately, what prevents this approach is that post-marker messages may be received on some connectors before all the pre-marker messages have been received. The resultant state would then incorporate not only pre-marker messages, but also any post-marker messages that arrived before the snapshot was taken.

The key to solving this problem is to find a way to take snapshots such that all pre-marker messages can be incorporated into the checkpointed state without inadvertently including the effects of post-marker messages in the checkpoint. A straight-
forward approach is to suspend the processing of post-marker messages by stalling the input on channels that have received a checkpoint marker until the same marker has been received on all channels. This effectively forces the checkpoint markers to appear as though they were received simultaneously. However, this means that certain computations will be stalled and performance of the application will suffer as a result. Ideally, this should not be necessary.

Yet another approach is for pebbles to take snapshots of a virtual state. This virtual state would appear as though all pre-marker messages arrived before any post-marker messages even though such a state may not have existed in the actual pebble execution. In this scheme, upon receiving a checkpoint marker for the first time, each pebble would take a snapshot of its state and set it aside as a virtual state. Any subsequent messages would then be acted upon based on whether they are pre-marker or post-marker messages with respect to their connector. While the pebble will act on all messages in the order that they arrive regardless of their type, only pre-marker messages will be allowed to affect the saved virtual state. The way the virtual state would be affected would be as though this state were the actual state of some virtual pebble, and that only pre-marker messages arrived at this virtual pebble. Once all connectors have relayed the marker, this indicates that there are no more pending pre-marker messages and the virtual state can then be sent to the checkpoint store.

This method ensures that the state recorded does not require any logging of messages, is consistent with that of the previous pebbles, and yet allows computations to continue during checkpointing. A downside of this is that the pebble needs to keep track of two sets of state during checkpointing. Should several checkpoint attempts be active in the system at the same time, each pebble may have to keep track of multiple sets of state. If one input connector had a much higher latency than another connector, the lag between the two connectors may very well span several different checkpoint markers with each marker necessitating a separate virtual state. Even if the latency of each input connector were exactly the same, the path from a root pebble to each connector may traverse a different number of pebbles and the arrival
of the same marker on two different connectors may still have a significant lag.

One can see that the number of state sets that each pebble would have to keep track of is approximately equal to the rate that checkpoint requests are being issued, multiplied by the time difference between the first and last receipt of the marker from different connectors. Hence, reducing the frequency of checkpoints is a straightforward way of reducing the number of state sets each pebble has to keep track of simultaneously. However, even with a reduced rate of checkpointing, another problem becomes apparent once we consider the virtual states of pebbles further down the pipeline.

In the original design, the receipt of checkpoint markers indicated when the sending pebble took a snapshot of its state and was a clear separator for which messages on the connector were sent before the snapshot and which were sent after the snapshot. In other words, pre-marker messages were the same as pre-snapshot messages and post-marker messages were the same as post-snapshot messages. Now with virtual state checkpointing, this distinction becomes less clear since the messages sent by the actual pebble will not necessarily be the same as what would have been sent by the virtual pebble corresponding to the recorded virtual state. Yet at the same time, in order for each marker recipient’s virtual state to be consistent with the marker sender’s virtual state, the marker recipient’s virtual state must be the state that would have existed if the recipient pebble had received messages from the virtual pebble, and not from the actual pebble.

If the output of a particular pebble is dependent only on its inputs, and not on its state, then the hypothetical outputs of its virtual pebbles would be a subset of the actual outputs. With this in mind, one could attach a header to each message that indicates whether or not the message should be incorporated into the virtual state. The receiving pebble could then use this header to decide which messages should be sent to its virtual pebble and which messages should be processed only by the actual pebble.

However, not all pebbles will produce output messages that are independent of their state. In fact, there is little point in checkpointing such pebbles to begin with.
since they are, in effect, memoryless. For all other pebbles with output affected by state, the outputs of their virtual pebbles may be completely different from the actual outputs since their recorded virtual state may not correspond to any actual state. In this case, the virtual pebble would need to do more than merely update its virtual state – it would also need to generate the corresponding hypothetical outputs to be sent to the recipient pebble’s own virtual pebble. With this additional requirement, the responsibilities of the virtual pebble would effectively grow to match those of an actual pebble and the presence of such virtual pebbles with their own channels among themselves would become like a new instantiation of the entire application.

Now, we see that the overhead of virtual state checkpointing has grown from having to keep track of multiple state sets in each pebble to having to create new instantiations of the application for the duration of each checkpoint. With a high rate of checkpointing, application resources would quickly become consumed by checkpointing alone. This leads us to reconsider the stalling approach to dealing with pebbles that have multiple input connectors.

Even though the stalling of input connectors can have a negative impact on performance, it turns out that this impact may not be that serious. When investigating the performance hit, it is useful to understand why pebbles may have multiple inputs in the first place. In some cases, a pebble may have multiple inputs because it performs a function that operates on several input values to produce a single output. Such pebbles would operate on their inputs synchronously and would have to stall if some inputs were not yet available. Since these pebbles stall anyway to ensure synchrony of inputs, checkpoint markers would also arrive simultaneously on all inputs and hence the stalling approach will have no additional performance penalty on such applications.

At the same time, some pebbles exist that do not require synchrony of inputs and can perform operations on inputs as and when they arrive. An example of this would be a pebble that serializes multiple input sources into a single output. In such cases, stalling of inputs would negatively impact performance and may result in an application that appears jerky since stalls would occur during each checkpoint.
However, one could attempt to lessen this effect by reducing the lag time between the receipt of the first and last marker for each checkpoint. This could be done by introducing a buffer that increases the latency of the lower latency channels such that checkpoint markers will arrive roughly in synchrony with each other. In fact, such buffers would already exist in the form of input buffers that are holding stalled messages. All that needs to be done is to tweak whether or not these buffers increase latency by specifying whether or not they should limit their rate of output when no stall is in progress. In a sense, such buffers will enable the stall problem to be self-correcting since the buffers would be forced to synchronize each time a stall occurs, thus minimizing the length of future stalls.

In this way, the stalling approach has a minimal impact on the speed and jerkiness of an application while providing a very simple solution to the problem of dealing with pebbles that have more than one input. Hence, the proposed framework adopts the stalling approach by offering the following modified version of Chandy and Lamport’s distributed snapshot algorithm:

1. upon receiving marker $m$ on channel $c$, pause channel $c$

2. if $m$ has been received on all input channels,
   1. save snapshot
   2. send marker $m$ on all output channels
   3. resume all input channels
   4. snapshot complete

**Cyclic Pebble Structures**

Even with the modified version of the checkpointing algorithm, there still exists a problem with application instantiations that have cyclic pebble structures. Consider to the pebble structure shown in Figure 4-5, where pebble $A$ outputs to pebble $B$, $B$ goes to $C$, $C$ goes to $D$, and $D$ leads back to $B$. Pebble $B$ has two inputs, one from pebble $A$, and one from pebble $D$. When a checkpoint is initiated at $A$, pebble
Figure 4-5: A simple application with a cyclic pebble structure.

B receives a checkpoint marker from A and stalls until the same marker is received from pebble D. However, D will not send a marker until it has received one from C, which in turn will not send one until it has received one from B. As a result, we see a deadlocked situation where pebble B is waiting for itself.

In order to prevent deadlock, pebble D could be told that the input from D is actually dependent on itself, and so it should not wait for input from D before propagating the checkpoint marker. Additionally it would ignore checkpoint messages from D so as to prevent the same checkpoint message from looping around the cycle infinitely.

In this way, the snapshot from B would be concurrent with that of C and C’s would be concurrent with that of D and hence B’s snapshot must be concurrent with D’s. However, between the time that B and D took their snapshots, several messages may have been sent from D to B and these would be omitted from the snapshot.

Let us refer again to Figure 4-4 to analyze this in a specific example. Here, A is connected to B, B is connected to C, and each of the pebbles outputs the sum of the current and previous inputs. Suppose we further connect C to A, we form a cyclic path. Even with a single data value flowing down the pipeline, we see that at all times, either some pebble is busy processing its input, or some pebble has finished processing its input and a message has been sent on its output connector. Thus, in order to take consistent snapshots of such a program instantiation, it is necessary to halt the cycle momentarily and log the message flowing from C back to A so
that the message does not alter $A$’s state and make it inconsistent with that of the other pebbles. Without logging messages, our checkpointing algorithm simply cannot support program instantiations with cyclic structures.

The reason messages could not be logged in the first place was that these messages may have no meaning in future reconfigurations of the same application. Now, supposing the planning engine identified one link in the cycle that would map to some link in all other possible configurations of the application, it would now be possible relax our restriction and log messages on this link since these messages would continue to have meaning even after a reconfiguration.

As an example, some numerical estimation problems involve loops where the last estimate is fed back into the estimation procedure and iteratively used make a more precise estimate. In this case, the similar link could be that which carries the last estimate back into the start of the procedure.

In this way, our checkpointing framework can thus support cyclic structures, but only if some link in the cycle is common for all possible instantiations of the application. It is the responsibility of the planning engine to recognize this, and it should inform the pebble on the receiving end of this link of the special input, $c_{sp}$. This pebble should then follow the following modified checkpointing algorithm upon the receipt of each marker:

1. upon receiving marker $m$ on channel $c$, pause channel $c$ if $c \neq c_{sp}$

2. if $c = c_{sp}$,
   1. stop logging of messages on $c_{sp}$
   2. snapshot complete

3. if $m$ has been received on all input channels except $c_{sp}$,
   1. save snapshot
   2. send marker $m$ on all output channels
   3. resume all input channels
Connectors on UDP Channels

Earlier it was stated that Chandy and Lamport’s algorithm relies on links between components having a FIFO nature. The algorithm uses checkpoint markers as a separator between messages that should be included in the checkpoint and those that should not.

When pebbles are linked with TCP channels, checkpoint markers perform their job since TCP guarantees in-order delivery. However, TCP is not always the best transport for all applications. Some applications make use of UDP connections which do not guarantee delivery of packets, and do not guarantee that packets delivered are in-order. For example, in sound and video applications, the reordering or loss of frames of video and bits of sound over a small time period is not noticeable to the human end-user. At the same time, relaxing the ordering constraint allows packets to be relayed more quickly.

Now since messages on a UDP channel can be reordered, the checkpoint marker can no longer serve its purpose as a separator and this is worrying because these markers were being relied upon to produce consistent checkpoints. However, it is our conjecture that since an unreliable channel is being used to join two pebbles, it probably does not matter whether these two pebbles are consistent with each other or not. After all, one pebble logging that it has sent a message on such a channel does not necessarily mean that it was eventually received on the other end.

As long as the checkpoint markers themselves still arrive, and arrive in-order, our checkpointing algorithm should still function albeit with some tolerable inconsistency between pebbles joined by UDP channels. Guaranteed, in-order delivery of checkpoint markers can be easily achieved through some sequence of acknowledgments over the UDP channel.
4.3 Improvements and Optimizations

Several small improvements and optimizations to the proposed framework are possible to make it perform better and these are presented here. For the sake of keeping the framework design simple to understand, none of these will be included in the framework specification laid out in the next chapter. Nevertheless, they will be briefly discussed here. Most of these are simple to implement and will not significantly affect the base specification of the checkpointing framework.

4.3.1 Initial Input Retention

After an unexpected failure of a pebble, a forced reconfiguration would have to take place and the new joining pebbles would inherit the last checkpointed state of the leaving pebbles. Additionally, all other pebbles would need to have their state reverted so that their state remains consistent with that of the new pebbles. If the root pebbles in the application instantiation graph were receiving input from some external sources, this would result in a loss of data.

Considering that a failure had occurred, this loss of data could be tolerable. However, it is possible that such a loss could be prevented. One way of doing this would be for all root pebbles in the structure to keep a log of messages that were received since the last checkpoint. After a reconfiguration, as long as the root pebbles did not fail, these logged messages could then be replayed.

4.3.2 Timelines

Inspecting an application instantiation graph, it might be possible for parts of the graph to be disconnected from each other. For example, a basic video conferencing application may have one subgraph of pebbles for sound, and another subgraph for video. If this simple application does not have any mechanism for synchronizing sound and video, the two subgraphs may not be connected at all.

In such applications, some of the sections may have long chains of pebbles and thus have a long checkpoint time, while the opposite could be true for the some of
Since pebbles in different sections do not actually have messages being passed between them, a snapshot from a pebble in one section is always concurrent with a snapshot from a pebble in another disconnected section. This is true regardless of when these two snapshots were taken relative to each other.

These two sections of pebbles effectively have different timelines, and thus one optimization could be made by checkpointing each of these timelines separately. One example of how this would be useful is if the pebbles on the shorter timeline required frequent checkpoints at a rate higher than what would be possible on the longer timeline.

Additionally, if a pebble were to fail and a forced reconfiguration was necessary, only the pebbles on the same timeline as the failed pebble would need to have their state reverted. Pebbles on all other timelines could continue regular operation.

4.3.3 Buffer Retention Behavior over Reconfigurations

During an intentional reconfiguration, only the subgraph of pebbles being replaced needs to have its execution suspended. During the reconfiguration, all other pebbles can continue regular operation. At the same time, some pebbles may have connectors that lead to pebbles that are being swapped out. In order for there to be no loss of data, pebbles should automatically pause and buffer such output connectors until reconfiguration has been completed. This is how the framework specification given in the next chapter behaves.

While this buffering prevents data loss over an intentional reconfiguration, it may not be the intended behavior for all applications. Earlier in Section 4.1.1, we gave the example of a user streaming live television onto his handheld computer. When an intentional reconfiguration takes place in such an application, the user may be interested only in receiving the most current video feed and the buffered frames would actually be unwanted.

The simple solution to this is that the application script should relay to the planning engine what its desired behavior during intentional reconfigurations is. Accord-
ingly, the planning engine should identify which connectors should and should not be buffered during a reconfiguration. The relevant pebbles should then instructed to discard output on those connectors during a reconfiguration.

A further modification in this direction is possible by having the relevant pebbles discard only portions of the output. In the example of the live television stream, the buffering could be set up such that only one out of every four frames were buffered to be replayed after the reconfiguration has taken place. After reconfiguration, playback of missed frames would then happen at four times the regular speed and the user could quickly run through what he missed before returning to the live feed.

4.4 Summary

Here we have gone through the user requirements that influenced our proposed checkpointing framework, as well as walked through the decision process behind some of the more important design choices.

The result of this was a system where checkpoints are triggered by pebbles themselves, and state is stored in a few central locations. Different checkpointing algorithms were chosen and modified for the different kinds of checkpoints required by our framework. Several possible optimizations to the recommended specification were also briefly discussed.

Given this understanding of the basic framework and the rationale behind many of its aspects, we are now ready to dive into the specification of the proposed semantic checkpointing framework.
Chapter 5

Design and Implementation

Keeping in mind the design considerations detailed in Chapter 4, a design specification was laid out. Using this specification, one can modify a standalone application to make it runtime-reconfigurable. Preferably, one can also implement this specification with a suitable planning and reconstituting engine to construct a general-purpose checkpointed execution environment for runtime-reconfigurable applications. The latter was attempted by implementing a prototype on top of the Pebbles System and this environment was tested with a simple chat application.

This chapter first lays out the design specification of the proposed checkpointing framework, then proceeds to give details and a discussion about the prototype implementation.

5.1 Roles of Each Component

The semantic checkpointing framework presented in this thesis is specified by the roles that each component will have to play in ensuring that global state can be captured and redeployed as necessary. Four key players are identified in this framework: the checkpoint stores, the pebbles, the reconstituting engine, and the planning engine.

In this section, we will describe the additional functionality required of each component as a set of exported remote procedure calls (RPCs). Of course some mechanism other than RPC could be employed to perform the same functions. After all,
many of the requests presented below do not require any reply and are simply messages passed between components for coordinating checkpoints. However, referring to such coordination messages as procedure calls is useful for explicitly describing the meaning of each these messages.

5.1.1 Checkpoint Store

Earlier we recommended that there should be one checkpoint store for each group of pebbles in the same local network. However, we shall view the checkpoint stores together as a single entity here for the sake of promoting a simple understanding of the basic checkpointing framework. With an understanding of how a single checkpoint store would operate, one can infer the roles that each checkpoint store must perform.

Together, the collective checkpoint store is responsible for initiating regular checkpoints among pebbles, initiating special checkpoints before a reconfiguration, logging application state from each of these checkpoints, and restoring state whenever the application is reconfigured.

In order to perform this function, the following procedure calls are specified:

setStructure(structureDesc): This function is called by the planning engine during the initialization of the application and is used to pass a description of the pebble structure, structureDesc to the checkpoint store.

This structure description provides information about each pebble in the initial application instantiation including its pebble ID, a specification of what state is to be checkpointed, and a listing of input and output connectors with their associated pebbles.

Each piece of state to be checkpointed is specified by a pair of keys: a local key for looking up state to be checkpointed within the pebble, and a global key that determines what the checkpointed value maps to within the global application state.

Upon receipt of this structure description, the checkpoint store should prepare to be contacted by individual pebbles as well as create an initial empty global
state in anticipation of receiving checkpoints later on.

This function should be called before the pebbles are initialized so that the checkpoint store can be ready to receive snapshots the moment the application has started.

\textbf{pebbleHello(\texttt{pebbleID, pebbleAddress})}: This function is called by each pebble after it is initialized. This provides the checkpoint store with the pebble's address so that the pebble can be contacted later on when necessary.

This function returns the state keys that this pebble should include in its snapshots as well as any initial state values each of these keys should have in the pebble. During an initial instantiation, this initial value would be empty. On the other hand, after a reconfiguration, this initial value will carry the checkpointed state.

This function will also return a flag stating whether the pebble should pause its inputs and outputs after initialization. At the initial instantiation, these flags will not be set since such behavior will be handled by the planning engine. They will, however, be set during a reconfiguration. During a reconfiguration, some pebbles may need to have their state reverted to some checkpointed state and messages from such pebbles should be ignored until the revert has finished. Additionally, new pebbles will be started and will not be ready to receive messages until their initialization is complete. The checkpoint store is informed when all this has happened and is at a position to resume inputs and outputs at that point of time.

\textbf{requestCheckpoint()}: In Section 4.2.2, it was mentioned that some pebble would be designated to determine when checkpoints are to take place. When such a pebble decides it is desirable for a checkpoint to be initiated, it sends a \textbf{requestCheckpoint} call to the checkpoint store.

The checkpoint store may either honor the request and initiate a checkpoint, or it may ignore the request if there are already too many checkpoints in progress.
in the application.

The checkpoint store initiates a checkpoint by first generating a unique timestamp, and then calling \texttt{initiateCheckpoint(timestamp)} at each of the root pebbles in the program structure. A root pebble is one which has no inputs from any other pebble in the program structure.

\texttt{putStateSet(pebbleID, timestamp, keysValues)}: This function is called by each pebble whenever it takes a state snapshot. \texttt{pebbleID} tells the checkpoint store which pebble this set of state belongs to, and \texttt{timestamp} carries the timestamp of the relevant checkpoint request. \texttt{keysValues} contains the actual set of state keys and values being checkpointed.

Once all the pebbles have called \texttt{putStateSet} with a particular timestamp, this indicates that the global checkpoint for that timestamp is complete. The checkpointing engine keeps a copy of the most recent global checkpoint, and this checkpoint is replaced with the newly completed checkpoint if it is has a newer timestamp.

\texttt{forcedSwap(newStructureDesc)}: After some pebble has failed unexpectedly, the planning engine will need to initiate a forced reconfiguration. This intent is communicated to the checkpoint store through the \texttt{forcedSwap} call.

During a forced reconfiguration, new pebbles may be instantiated to join the pebble structure and these pebbles will adopt the last checkpointed state. At the same time, old pebbles remaining in the structure will also have to have their state reverted to this checkpoint so that their state remains consistent with that of the new pebbles.

The checkpoint store's role in the procedure is to transfer checkpointed state to all pebbles concerned. Each of the new pebbles will call \texttt{pebbleHello} and the checkpointed state can be transferred to it at this stage. Since old pebbles remaining in the structure will not be re-instantiated, they will not call \texttt{pebbleHello}. Hence, the checkpoint store will update the state of these pebbles
by calling \texttt{reHi(keys, keysValues, pauseInputs, pauseOutputs)} on these pebbles, where the arguments of this call are the same as what would have been returned by \texttt{pebbleHello}.

After all the new joining pebbles have called \texttt{pebbleHello}, and after \texttt{reHi} has been called on the remaining pebbles, the state of all the pebbles becomes consistent with each other and their input and output connectors can be resumed. The checkpoint store now calls \texttt{resumeInput()} on all pebbles, followed by \texttt{resumeOutput()} on all pebbles.

\texttt{intentionalSwap(newStructureDesc)}: Occasionally, the planning engine may perform an intentional reconfiguration when conditions change such that it is desirable to do so. Before actually performing the reconfiguration, this function is called. The checkpoint store then performs a special checkpoint so that there is no data loss over the intentional reconfiguration.

The checkpoint store carries out the intentional reconfiguration by sending a \texttt{initiateSpecialCheckpoint} call to all leaving pebbles, as well as by calling \texttt{finishConnectors} on all retained pebbles that have output connectors to some leaving pebble. These pebbles will act accordingly to carry out the special checkpoint.

After all the leaving pebbles have called \texttt{putStateSet}, this indicates that they have all taken snapshots and the special checkpoint is complete. The checkpoint store then calls \texttt{spComplete} on the planning engine and the planning engine then proceeds with the reconfiguring of pebbles.

The checkpoint store then proceeds as it would have in \texttt{forcedSwap}, except that no pebbles need to have their state reverted. Additionally, not all pebbles will have their input and output connectors paused and hence \texttt{resumeInput()} and \texttt{resumeOutput()} need only to be called on relevant pebbles. These pebbles include new pebbles joining the program structure, as well as retained pebbles that had output connectors to leaving pebbles. The reason for the latter will be apparent later when we describe the steps that each pebble would take to
perform a special checkpoint.

5.1.2 Pebbles

Pebbles are the building blocks of runtime-reconfigurable applications in our framework and are holders of all the state we are interested in checkpointing. Pebbles must thus have some interface for sending snapshots of state to the checkpoint store, as well as for accepting new state to be incorporated in the event of a revert to an older checkpoint. Additionally, they must be able to coordinate snapshots among themselves so that the resultant global checkpoints are always consistent.

Pebbles will store several bits of state, however, the planning engine may be interested in those that are semantically significant in the context of the entire application. How this is specified is through a list of state keys that will map to some global key in the application checkpoint. This allows state to be carried forth over reconfigurations even if the pebbles involved have different keys for values with the same meaning.

In our checkpointing framework, pebbles must export the following additional functions:

\text{init(}\text{pebbleID, cpStoreAddress})\text{: When the planning engine instantiates the pebble, it should additionally inform it of its own pebble ID, as well as the address of the checkpoint store.}

After the pebble has finished internal initialization, it will then call the \text{pebbleHello} function in the checkpoint store to receive the keys that it should checkpoint, optional initial values for these state keys, as well as flags determining whether its input and output connectors should be paused initially. The set of keys is stored internally and determines the contents of the snapshots that will be sent to the checkpoint store later on.

\text{reHi(}\text{keys, keysValues, pauseInput, pauseOutput})\text{: After a forced reconfiguration, pebbles that are retained across the reconfiguration would need to have their state reverted to the last checkpoint. Since these pebbles have not been...}
newly instantiated, they will not call `pebbleHello` on the checkpoint store. Instead, it is the duty of the checkpoint store to call `reHi` on each retained pebble where the arguments are the same as the return values of a `pebbleHello` call.

Upon receipt of this call, the pebble resets its state to the received state and optionally pauses its input and output connectors awaiting future instructions from the checkpoint store.

**initiateCheckpoint**(`timestamp`): The function is called by the checkpoint store on each pebble that does not have input connectors from another pebble and is used to initiate a regular checkpoint from this pebble. The checkpoint algorithm used for this was described in Section 4.2.4.

This initiating pebble has no input connectors and hence does not need to wait for markers before proceeding with the snapshot. Instead, it waits till any in-progress transactions have completed, then pauses transactions so that it can make a snapshot of stable state. This snapshot is sent to the checkpoint store via the `putStateSet` call and is accompanied by the timestamp of the related checkpoint.

Next, the pebble sends out a checkpoint marker on each of its output connectors. This checkpoint marker is differentiated from regular data through a header and is accompanied by the timestamp of this checkpoint.

After this, the pebble resumes normal operation.

**initiateSpecialCheckpoint**(`timestamp`, `leavingPebbleIDs`): This function is called by the checkpoint store on each leaving pebble before an intentional reconfiguration.

Upon receipt of this call, the pebble should stop any transactions that do not rely on input messages. It should then process all of the incoming messages until the input connector has been closed (perhaps signaled with a final packet). Once all inputs have been consumed, processed and all outputs flushed, the pebble has reached a stable state and is ready to take its snapshot. The pebble first
closes its output connectors so that pebbles downstream know that no more
data will be received. Finally, the pebble calls putStateSet on the checkpoint
store then stops operation in anticipation of being destroyed.

finishConnectors(connectorIDs): During a special checkpoint, leaving pebbles
will wait for their input connectors to be closed as an indicator that the input
queues have been emptied. Since some of the leaving pebbles will have inputs
from pebbles that are not leaving, these inputs will also need to be closed.

When this function is called, the pebble should buffer outputs on the specified
connectors and send a final packet over the connectors before closing them.
After the intentional reconfiguration, the reconstituting engine will reconnect
this output connector to some other pebble but data flow will not resume until
it is instructed to do so by the checkpoint store. This prevents data from being
sent to a pebble that has not yet finished initialization.

resumeInput(), resumeOutput(): These functions are used by the checkpoint
store to resume inputs and outputs that may have been paused by a pebbleHello,
reHi, or finishConnectors. Note that when input connectors are paused, all
inputs received should be discarded. When output connectors are paused, any
data generated by the pebble should be buffered for sending in the future.

5.1.3 Reconstituting Engine

While the reconstituting engine does not play any additional role in the recommended
checkpointing framework, it is useful to mention what is required of it for reconfigu-
trations to be possible. At a minimum, it should provide methods for instantiating a
pebble, destroying a pebble, connecting two pebbles, and disconnecting a connector.

5.1.4 Planning Engine

The role of the planning engine is to instantiate the initial program structure, rec-
ognize when reconfigurations are desirable or necessary, and coordinate the various
components to carry these reconfigurations out. The tasks it needs to perform in order to fulfill these roles are best described through a series of scenarios which will be provided in the next section. It does, however, need to provide one external call for the purpose of checkpointing. This is the spComplete call that indicates that a special checkpoint has been complete, and that the planning engine can proceed with rearranging the pebbles.

5.2 Scenarios

How everything fits together is best understood by going through the steps of the following basic scenarios: application instantiation, regular checkpointing, forced reconfigurations, and intentional reconfigurations. These are all the possible situations that an application can be in with respect to checkpointing.

5.2.1 Initialization

1. At program startup, the planning engine reads the application script and decides on the best configuration for the application at that time. At this time, a suitable checkpoint initiator is chosen from among the pebbles.

2. It also decides what sets of state are semantically significant in the context of the entire application and maps these to local state keys in each of the chosen pebbles.

3. The planning engine then instantiates a checkpoint store for this application and sends it the initial program structure.

4. The planning engine calls the reconstituting engine to initialize and connect each of the pebbles in the program structure.

5. As each pebble is initialized, it sends a pebbleHello call to the checkpoint store to inform it of its address.

6. The pebbleHello call returns with a list of state keys of interest.
7. Application is ready for normal operation.

5.2.2 Regular Checkpoints

1. At various points during normal operation, the designated checkpoint initiator pebble may decide that it is desirable to have a checkpoint taken and sends a requestCheckpoint call to the checkpoint store.

2. Checkpoint store chooses whether to honor this request. If the request is to be honored, it is done by generating a timestamp and sending a initiateCheckpoint call to each of the root pebbles in the pebble graph.

3. Each root pebble pauses operation to send a snapshot to the checkpoint store using putStateSet, propagates a checkpoint marker across its output connectors, then resumes normal operation.

4. Once each pebble has received checkpoint markers on all of its incoming connectors, it sends its snapshot to the checkpoint store via a putStateSet call, propagates the same checkpoint marker across its own outputs, then resumes normal operation.

5. When every pebble has responded to the checkpoint store, this indicates that the checkpoint is finished.

5.2.3 Forced Reconfiguration

1. The planning engine recognizes the need for a forced reconfiguration and plans a new configuration for the application. Global state keys are remapped across each pebble and a new checkpoint initiating pebble is chosen.

2. The new structure is sent to the checkpoint store via the forcedSwap call.

3. The checkpoint store analyses the new structure and redistributes state as necessary. It then calls reHi on each of the pebbles retained over the reconfiguration
to reset their state to that of the last checkpoint, and to pause all of their inputs and outputs.

4. The planning engine calls the reconstituting engine to instantiate each of the new pebbles.

5. As each new pebble is initialized, it calls pebbleHello on the checkpoint store and receives its initial state. It is also instructed to pause all inputs and outputs.

6. After all new pebbles have responded to the checkpoint store, this indicates that state of all the pebbles is consistent and data flow can resume. All inputs are first resumed, followed by all outputs through the resumeInput and resumeOutput calls.

7. The planning engine instructs the reconstituting engine to destroy pebbles that have left the structure and the application continues regular operation.

5.2.4 Intentional Reconfiguration

1. The planning engine decides when an intentional reconfiguration is desirable and decides on the new configuration as above.

2. The planning engine sends the new configuration to the checkpoint store via a intentionalSwap call and awaits a spComplete call before proceeding with the actual reconstituting.

3. The checkpoint store calls initiateSpecialCheckpoint on each of the leaving pebbles and calls finishConnectors on each of the retained pebbles with outputs to leaving pebbles.

4. Each of the leaving pebbles processes all input until its input connectors have been emptied and disconnected then sends snapshot data to the checkpoint store with a putStateSet call.
5. When all leaving pebbles have responded to the checkpoint store, this indicates that the special checkpoint is complete. The checkpoint store sends a spComplete call to the planning engine and awaits pebbleHellos from new pebbles.

6. The planning engine uses the reconstituting engine to instantiate new pebbles and connect them to the rest of the program.

7. As each of these pebbles is instantiated, it will call pebbleHello on the checkpoint store to receive initial state. All connectors will also be paused.

8. When all new pebbles have called pebbleHello, the reconfiguration is complete and all inputs can be resumed, followed by the resuming of all outputs.

5.3 Prototype Implementation

A prototype implementation was implemented on top of an early version of the Pebbles System developed by the Computer Architecture Group at LCS [15].

The checkpoint store was implemented as a standalone application based on the design specification given earlier. XML-RPC calls were chosen for communication between pebbles and the checkpoint store since this was how calls within the Pebbles System were implemented. Interestingly, implementing the checkpoint store as a standalone component utilizing XML-RPC calls allows this same checkpoint store to be used with different kinds of software components implemented in different languages as long as they adhere to the same specification.

One caveat of using RPC calls to perform the calls laid out in the specification is that deadlock is possible if not implemented carefully. For example, when a checkpoint initiating pebble requests a checkpoint through requestCheckpoint, if the checkpoint store grants this request it will proceed to call each of the root pebbles. Now, if the requestCheckpoint call did not return until this was done, and the initiating pebble was a root pebble, then a condition would result where the pebble cannot accept the initiateCheckpoint call from the checkpoint store since
it is waiting for the requestCheckpoint call to return. This situation was prevented by having the requestCheckpoint call return earlier on, and performing the actual checkpoint initiation in another function that would be called only after the return of the requestCheckpoint call. Other possible deadlock situations were prevented in a similar manner.

In order to make it easy to write a checkpointed pebble, a basic template for checkpointed pebbles was written. Pebble developers could then extend on this template for implementing their own pebbles. Admittedly, it would have been cleaner to modify the base pebble classes in the Pebble System implementation. However, this approach was not taken since the Pebble System is still under active development and would have been a moving target. Instead, the checkpointed pebble template was written such that the additional functions could be easily incorporated into the base class at some later stage.

A notable feature of the checkpointed pebble template is the use of transaction locks. Whenever a pebble using this template wishes to carry out a transaction, it should request a transaction lock. As long as there are no snapshots that need to be taken, this lock will be granted. After the transaction is completed, the pebble should release the lock. This requesting and releasing of transaction locks is the only additional function that a pebble would need to implement if using the template described here as the rest of the checkpointing duties are already taken care of in the template. In the template, once all checkpoint markers have been received and the pebble is ready to take a state snapshot, new transaction locks will not be granted and the snapshot will be taken only when the number of active transaction locks drops to zero.

A few modifications to the Pebbles System were necessary for implementing a functioning prototype. In particular, the Pebbles System was designed for a single instantiation of an application and, at the time of implementation of this prototype, did not export any obvious mechanism for destroying Pebbles or for disconnecting a connector. Additionally, the default failure mode when a connection was broken was for the application to stop running. This was not desirable since our environment
relies on pebbles joining and leaving the program structure while the application is running. Crude modifications were made such as masking connector failures even though the developers of the Pebble System may have some better approach in mind.

5.4 Test Application

In order to demonstrate the functionality of the prototype implementation, a simple application was written within the prototype environment. This was a chat application built by installing a text source pebble and a text sink pebble on each of the machines participating in the chat. These pebbles are shown in Figure 5-1. The state that was to be checkpointed was the text buffer displayed in the text sink.

![Screenshot from text source and text sink pebbles.](image)

Figure 5-1: Screenshot from text source and text sink pebbles.

Since the Pebbles and the Goals systems were not yet integrated at the time of writing, the planning engine was simulated with a dummy engine that would switch
between two hard-coded configurations. This switch could be chosen to be an intentional reconfiguration or a forced reconfiguration depending on which button was pressed.

The two different configurations were chosen to simulate a chat user moving between two different machines over the duration of that chat. The user may anticipate this move and this would result in a intentional reconfiguration. On the other hand, the user's computer may fail, necessitating a move to a different machine. This situation was simulated by killing the pebble processes on that machine and initiating a forced reconfiguration. As the text source pebbles and text sink pebbles are moved around, capturing and redeploying the text buffer allowed the entire history of the chat to be displayed even after a reconfiguration.

For both these kinds of situations, the application behaved correctly. When a forced reconfiguration was triggered, the history buffer was only restored to the last buffer and some parts of the conversation were lost. On the other hand, the intentional reconfiguration managed to capture the entire conversation.

In this simple application, regular checkpoints took an average of 0.7 seconds between the sending for the first marker and receipt of the last snapshot. Special checkpoints took a similar average of 0.9 seconds.

In contrast, forced reconfigurations took an average of 3.9 seconds and intentional reconfigurations took an average of 5.0 seconds. Most of this time was spent initializing new pebbles and drawing their windows on the screen. Intentional reconfigurations took a little longer than forced reconfigurations because of their need to perform a checkpoint just before switchover. Observing this, it may be possible to reduce the perceived reconfiguration time of intentional reconfigurations by allowing the old set of pebbles to maintain control of the application until the new set of pebbles has started up. The special checkpoint can then be followed immediately by switchover of control, dropping the perceived reconfiguration time to a little over the special checkpoint time.

An issue cropped up with one test when a connection failure between pebbles resulted in a checkpoint marker going missing. Since there was no mechanism imple-
mented to deal with this, the intended recipient of that checkpoint marker was stuck waiting for the checkpoint marker and no more checkpoints could be completed. Some mechanism should have been put in place for a pebble to abort a checkpoint if a marker is not received after the timeout. This abort could then be signaled to the checkpoint store and the checkpoint store could subsequently instruct all pebbles to abort that checkpoint.

Admittedly, this simple application is not the best demonstration of the abilities of the checkpointing framework. A better example would have been one where the reconfiguration happened between two significantly different program structures. For example, a multi-user chat application could switch between a point-to-point communication model, and a model with a central chat server. Alternatively, the chat user interface could be switched between one where each participant’s messages appeared in a separate window, and one where all incoming messages were displayed in a single window. Both of these examples are possible with the semantic checkpointing framework presented here. Nevertheless, the simple test application that was built still served as a proof of concept in demonstrating that runtime-reconfigurable applications could be enabled through the use of semantic checkpointing.

5.5 Summary

The design specification of our recommended checkpointing framework outlined the roles that the checkpoint store, the pebbles, and the planning engine should play in order to coordinate checkpoints and ensure that application state can be effectively captured and redeployed as necessary.

This specification was implemented on top of the existing Pebbles infrastructure and successfully demonstrated the feasibility of using semantic checkpointing to enable runtime-reconfigurable applications. At the same time, several observations were made about the prototype implementation that prompted suggestions for improving the basic framework.
Chapter 6

Conclusions and Future Work

When this project was first started, it was thought that the design would simply involve keeping a dictionary of state, and employing some standard distributed checkpointing algorithm for keeping this state updated. Instead, it turned out that semantically carrying state from one program structure over to a significantly different one posed some tricky issues for the steps that a checkpointing algorithm should take. Nevertheless, a suitable solution was found that was at the same time straightforward to implement.

Through this thesis, we have shown that the semantic checkpointing of components is a feasible way of allowing applications to automatically maintain state over reconfigurations. Through our prototype implementation, we also demonstrated a programming environment that allowed one to easily build runtime-reconfigurable applications. In this environment, all checkpointing and reconfiguration was performed on behalf of the application, the only additional burden to the developer was that of requesting locks for each transaction. With the eventual integration of the Goals resolution engine into this environment, end-users will additionally be endowed with the ability for their applications to automatically and dynamically adapt to changes in their operating environment.

At the same time, such functionality is unlikely to come without its own share of trade-offs, and this is true in our approach. First of all, users will experience some reduction in speed both during checkpointing, and across a reconfiguration. While we
have attempted to craft an optimal algorithm for regular checkpoints, the time taken is still significant when compared with an uncheckpointed application where this takes no time at all. Furthermore, regular checkpoints are taken to allow partial recovery from failures, and certainly not all applications that crave adaptability also require this degree of reliability. In the case of intentional reconfigurations, an interruption is caused even though the user may have been satisfied with the previous configuration.

The other main trade-off we have identified is the possible limiting of applications. The first way this occurs is through the use of buffering when checkpointing components with multiple inputs. By buffering the input channels which relay checkpoint markers earlier, the ordering of inputs from different channels becomes modified with respect to their actual order of arrival. It is possible that this is undesirable for certain applications. The other limitation that occurs is in the choice of a programming model where applications are specified by goals and built by the combination of basic components. While this model makes it easier to develop applications that are runtime-reconfigurable, the fact that it differs from the norm may hinder its acceptance.

Even if these trade-offs prove to be overwhelming, this thesis still serves as a guide for developers intending to incorporate checkpointing into their own applications. It provides a sample model for the roles that key components should play, as well as points out important considerations for correctness when transferring checkpoints across application reconfigurations.

With regards to future work, three main tasks are identified. Firstly, the actual performance of this checkpointing framework needs to be optimized as well as quantified. The initial implementation here served more to convey an idea rather than to provide its final answer, and much can be done in the way of optimizations. With a more accurate idea of the performance cost of checkpointing, developers can also have a better gauge of the suitability of checkpointing for their purposes.

The second area of interest is in extending the flexibility of the checkpointing environment. In this first version, the number of connectors from the same pebble could not grow or shrink. Additionally, the state semantics were captured as a simple
one-to-one mapping from global keys to local keys. In the future, the environment should also support mapping of a single global key to state distributed across multiple local keys, or vice versa. With some translation, state could also be captured and restored across components that have different internal representations, as long as their values are semantically identical.

Finally, more applications need to be developed within this checkpointing environment in order to gain a better understanding of how flexible, or how limiting it truly is. Perhaps the best way of doing this is to integrate the Goals resolution engine into the environment, and release it as a programming environment to a community of users and developers. While the buffering of inputs and the programming model chosen may prove limiting in some areas, whole new types of applications could well flourish.
Bibliography


