An Investigation of Semantic Checkpointing in Dynamically Reconfigurable Applications

by

Hooria Komal

S.B., EECS M.I.T., 2004

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

January 2008

Author [illegible]
Department of Electrical Engineering and Computer Science
January 18, 2008

Certified by [illegible]
Dr. Karen Sollins
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

An Investigation of Semantic Checkpointing in

Dynamically Reconfigurable Applications

by

Hooria Komal

Submitted to the
Department of Electrical Engineering and Computer Science

January 18th, 2008

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

As convergence continues to blur lines between different hardware and software platforms, distributed systems are becoming increasingly important. With this rise in the use of such systems, the ability to provide continuous functionality to the user under varying conditions and across different platforms becomes important.

This thesis investigates the issues that must be dealt with in implementing a system performing dynamic reconfiguration by enabling semantic checkpointing: capturing application-level state. The reconfiguration under question is not restricted to replacement of a few modules or moving modules from one machine to another. The individual components can be completely changed as long as the configuration provides identical higher-level functionality. As part of this thesis work, I have implemented a model application that supports semantic checkpointing and is capable of functioning even after a reconfiguration. This application will serve as an illustrative example of a system that provides these capabilities and demonstrates various types of reconfigurations. In addition, the application allows easy understanding and investigation of the issues involved in providing application-level checkpointing and reconfiguration support. While the details of the checkpointing and reconfiguration will be application specific, there are many common features that all runtime reconfigurable applications share. The aim of this thesis is to use this application implementation to highlight these common features. Furthermore, I use our approach as a means to derive more general guidelines and strategies that could be combined to create a complete framework for enabling such behavior.

Thesis Supervisor: Dr. Karen Sollins
Title: Research Scientist, MIT Computer Science and Artificial Intelligence Laboratory

Acknowledgements

I would like to thank my advisor Dr. Karen Sollins for her constant support and encouragement throughout my time as a graduate student at MIT. Her enlightening and motivating discussions have provided me with a more insightful view of computer systems and have shaped my future research directions.

Table of Contents

List of Figures

Chapter 1

Introduction

Imagine the following scenario. A user "Jane" is playing an online game on her mobile phone while walking from one building to another. She walks into a building wherein her phone has low signal reception so the game slows down, or even worse comes to a complete stop. This new location, however, has a high-speed wired connection and a desktop computer with a large LCD display. Given these conditions, Jane would ideally like to move the game to the desktop computer and continue playing without any disruptions, delay, or loss of continuity. In practice such a switch will inevitably involve manual intervention, and hence cannot happen seamlessly. However, it can be stated with reasonable certainty that as the next best alternative, Jane would like to minimize the disruption and begin playing the game on the new computer without having to completely restart it.

For the user to switch from the mobile phone running an embedded operating system to another computer equipped with both a different operating system and completely different hardware, the application must be equipped with appropriate functionality to facilitate the switch. To enable such a move, the application requires support for a complete reconfiguration- it must provide the same high-level functionality even if the components, the modularization of the components or the platform changes. Being able to provide the same functionality with different configurations is just one part of the story. Given such a capability, Jane could start playing the same game on the computer but would still be unable to transfer the game state from her on-going game on the phone. In essence, the game would have to be restarted for Jane to switch to the other machine. While this is better than her being unable to play after the location change, there is room for tremendous improvement. In order for Jane to keep playing the game with an appearance of continuity, the application must support another functionality in addition to the ability to reconfigure - the checkpointing capability. For this game, or any other application, to continue functioning seamlessly under varying conditions it must support checkpointing and reconfiguration capabilities.

Though many games and applications possess the ability to checkpoint, the current implementations are far from capable of achieving the more general objectives stated above. For instance, many video games automatically save game progress periodically. This functionality is made possible due to the predictable, linear format of the games. Despite this simplified structure, the implementations are tied to the specifics of the platform and would not be able to address the example mentioned above.

1.1 Dynamic Reconfiguration and Semantic Checkpointing

Dynamic Reconfiguration refers to changing the configuration of an application while it is executing. Such a configuration change could involve moving a module from one machine to another, updating the application, or any other modification to the system while it is still running. Since the aim is to make these changes without stopping the application, the application has to continue providing the same high-level functionality it would have provided in the absence of that change. Such changes could be necessitated by changes in external conditions, e.g. available resources such as network bandwidth or by changes in user requirements. Dynamic reconfiguration is most useful for long-running and mobile applications, and with additional redundancy is also well suited to mission-critical systems. The reconfiguration should be handled in a manner such that the user does not notice any change in functionality except possibly a momentary disruption. In order to provide a seamless view to the user, any new configuration that results from this process cannot be independent of the previous state of the application. Thus a natural issue that is coupled with reconfiguration is the issue of capturing the state of the system and passing it on to a new configuration such that the application has the appearance of continuity even after configuration changes. This capturing of state is what we call semantic checkpointing. The term highlights the fact that the checkpointing has to occur at some high level of abstraction since the underlying components/organization may be very different at different points in time.

1.2 Problem Space Overview

The problem space dealt with in this thesis is centered on reconfiguration of distributed applications. The primary focus is on applications that continue to function even under varying conditions. Such varying conditions could occur because of the mobile nature of the application, the long-lived aspect of the application, changes to mask failures such as resource unavailability, or changes in control policy for resources. For the sake of discourse, the simplest class of applications to use as an example is mobile applications. As mobile users move from one location to another, resources such as network bandwidth and display devices could change dramatically. Having to halt and restart the application each time such an event occurred would be cumbersome and in the event of resource unavailability might result in loss of useful data. Additionally, the ability to reach a remote server with greater computational ability will vary with available network bandwidth.

Semantic checkpointing and reconfiguration, when appropriately performed, enable systems to adapt to such changing conditions without losing significant state. Mobile applications are not the only applications capable of benefiting from such an approach. In the next chapter, additional examples are provided to motivate the problem.

1.3 Thesis Overview

This thesis investigates the issues that must be dealt with in implementing a system capable of performing dynamic reconfiguration by capturing application-level state. The reconfiguration under question is not restricted to replacement of a few modules or simply moving modules from one machine to another. The individual components can be completely changed as long as the configuration provides identical higher-level functionality. As part of this thesis work, I have implemented a prototype application that supports semantic checkpointing and is capable of functioning even after a reconfiguration. This application will serve as an example of a system that provides these capabilities and demonstrates various types of reconfigurations. In addition, the application allows easy understanding and investigation of the issues involved in providing application level checkpointing (also referred to as semantic checkpointing), and reconfiguration support. While the details of the checkpointing and reconfiguration will be application specific, there are common features that all runtime reconfigurable applications share. My aim is to use this application implementation to highlight these common features. Furthermore, I use our particular approach as a means to derive more general guidelines and strategies that could be combined to create a complete framework for enabling such behavior.

One idea is worth mentioning at this point: most of the work done in the area of checkpointing to date, has focused on capturing low-level details of the execution environment. The current work adopts a different approach by focusing on application level checkpointing-capturing the details of the application from a high-level functional perspective. As mentioned before in the context of video games, most approaches to checkpointing are completely tied to the particular platform, and allow little if any cross-platform support. This approach is analogous to early programming languages: a new compiler had to be written for each platform; thus severely limiting portability and cross-platform use. Just as modern languages have separated high-level language details and platform specifics (e.g. the Java Virtual Machine, and the use of Microsoft Intermediate Language in the .NET framework), this thesis aims to separate the high-level application details critical to semantic checkpointing from the details of the underlying hardware/OS configuration.

1.4 Thesis Outline

This chapter serves as an introduction to the thesis. Chapter Two details the problem space dealt with and the motivations for dealing with this problem. Chapter Three performs a survey of existing work related to checkpointing, application reconfiguration and dynamic module replacement, showing both the strides that have been made, while highlighting the many opportunities that still remain. Chapter Four describes the sample application at the heart of these investigations, the reasons for choosing this particular application, and why it serves as a good example. Chapter Five discusses the particular approach to semantic checkpointing and reconfiguration in the context of the prototype application, and the issues I came across in these studies, the solutions employed and the lessons learned. Finally, Chapter 6 provides a summary of results along with future directions.

Chapter 2

Motivation and Problem Space

2.1 Motivation

The problem space being dealt with is centered on dynamic reconfiguration of distributed systems. For the sake of completeness, this section briefly describes the key characteristics of the type of systems relevant to this thesis. Distributed systems usually consist of a collection of computers connected through the network. The computers could be running different hardware and operating systems but through the use of middleware, they can coordinate their activities and share their resources giving users the perception of a single, homogenous computing facility. In addition, such systems are associated with concurrency, openness and scalability. Distributed systems have become increasingly popular because of the flexibility and power offered by combining resources of multiple computers to perform computational tasks in a scalable, concurrent and fault tolerant fashion.

While there are many advantages to such a set up, the flexibility comes at the price of increased complexity. Because there are multiple points of control, there may also be multiple points of failure. Adding networked components introduces a whole range of potential failures that would not be an issue for an application running locally on a single computer. Furthermore, developing a system composed of heterogeneous hardware and software components provides additional complications in presenting a transparent, homogeneous view to the user.

Within distributed applications, those that would benefit most from checkpointing and reconfiguration include mobile or long-running applications. There can be several reasons to modify such an application while it is running, necessitating the use of checkpoints to preserve state. Besides scheduled maintenance and upgrades, an application could need to be modified because of changes occurring in the runtime environment. Such changes in the runtime environment of the application could be a result of several issues such as the mobile nature of the application, the long-lived aspect of the application, changes to mask failures such as resource unavailability, or changes in control policy for resources. For these and other reasons, it has been established that there is a need to perform dynamic reconfiguration of the types of applications mentioned above.

There is a continuum of approaches to reconfiguration used in common applications, trading continuity for simplicity. The simplest approach to modifying or updating applications during execution is to take the application offline, perform the desired operation and bring it back up. While the user may be willing to tolerate a momentary disruption, they would not tolerate losing the application state completely and starting over. For applications that continue to operate in the same configuration and on the same platform (hardware and operating system), existing checkpoint and logging techniques can be used to reconstruct the application state before the reconfiguration or failure. However, if the application needs to be moved to a different platform or needs to be started in a different configuration such techniques may not suffice. This different configuration could be a partial or complete change in the modularization of the application- the underlying components after the configuration could be completely different from the original ones. An example of this configuration change is provided by our example scenario, explained in later chapters, in which the application changes from a server-centric architecture to a peer-to-peer structure while continuing to provide the same high-level functionality. Any checkpoints or logs taken using existing techniques include information that assumes a particular modularization of the application or the existence of a particular execution environment. Such checkpoints are insufficient to initialize a new modularization of the application.

For the purpose of discussion, mobile applications provide the simplest and most illustrative platform on which to develop new checkpointing and reconfiguration approaches. As mobile users move from one location to another, resources such as network bandwidth and display devices could change dramatically. Having to restart the application each time such an event occurred would be cumbersome and in the event of resource unavailability might result in loss of useful data. The first chapter described a particular scenario in which a user moves to a new location, with superior hardware, while playing an online game. Given the change in resource availability, the savvy user might like to move the game from a small-screened cell phone to a more powerful desktop with a larger display, but without having to completely restart the game.

Semantic checkpointing, if performed properly, would enable applications to adapt to such changing conditions without significant loss of state. Mobile applications are only one group among many that would benefit from such an approach. As another example, consider a very computationally-intensive task being split up across multiple computers. Having the ability to dynamically reconfigure in this scenario would be very beneficial: if a new machine were added to the cluster, the application would be able to make use of the new machine and take advantage of the increased hardware immediately and dynamically. In addition, if this or another machine were to fail, the application could roll back to the most recent checkpointed state instead of having to start from scratch. Ideally, this would happen automatically, preserving the appearance of continuity and requiring no manual intervention.

2.2 Project Scope

The goal of this thesis is to explore and identify the key issues involved in designing and implementing an application supporting semantic checkpointing and reconfiguration. The particular type of reconfiguration under focus involves a complete rearrangement of the modules involved in the application. The components of the application can completely change between different configurations with only one constraint being satisfied: the high level functionality of the application remains the same. The broad questions I am seeking to answer are as follows. In a system that can be stopped and restarted with a completely different set of components, what are the key issues involved in checkpointing and what capabilities do we require to enable dynamic reconfiguration? Since we are not constraining ourselves to module-level changes and/or replacements, the system state cannot be captured in a manner that depends on the particular design or implementation of the module. As will be explained in additional detail later, the mechanism proposed in this thesis focuses instead on semantic checkpointing- capturing state in a manner that is meaningful at some high-level abstraction. A checkpoint captured in terms of such an abstract state can be used to reconstitute the high level functionality even with a completely different set of modules.

2.2.1 Prototype application overview

Applications designed for mobile phones are particularly likely to exhibit the need for such a rearrangement. With that in mind, I have designed a prototype application that exemplifies the type of application users may run on their desktops as well as their cellular phones. The application I have designed is a simple game of Tic Tac Toe meant to be played by two users. This particular application was chosen as a simple example that can demonstrate the need and utility for the complete rearrangement of the application components. The details of the component architecture will be described later but a brief overview is presented here.

The Tic Tac Toe application can run under two different configurations. The first configuration is based on the normal client-server model: each user runs the client application on their machine or cellular phone and the game is played through a central server. The central server, which controls the game logic, is offered as a JINI service. By making the game server available as a JINI service, it can be made independent of any physical device. Java's JINI technology, which will be described later, offers mechanisms that the clients can use to search for and locate the game server without having to worry about low-level communication details. When the client components start up, they need not know the physical location of the game server; the game server can be started on any machine that has access to the network and advertises the interface. The next step involves a join protocol wherein the clients establish a communication channel with the game server and express an interest in playing a game. Once the game server has messages from two clients, it starts a game and notifies the two players involved. The game server coordinates the game logic and keeps track of the state of the game including the state of the board and turns of the players. Both users play and watch the progress of the game through a GUI that is part of the client component. This user interface is continuously updated to reflect the state of the game as the clients query the game server. Besides all the modules making up the client component and the game server, there is an additional component involved in this configuration. This additional component is a checkpoint server which can be delegated the task of storing and retrieving checkpointed states.

The second model, a peer-to-peer configuration, does not rely as much on a central server. In that model, two users interested in playing a game directly contact each other. The initial rendezvous of the two clients is coordinated through a central server but once each user knows their opponent, the game can be played between the two players directly. It is not necessary to remove the central server completely in this model. The idea is simply to have the game logic be handled by the client itself to simulate a situation where the client machine is powerful enough that the user prefers to perform all computation locally. Because the client components are responsible for controlling and coordinating the game, it is natural that the modules involved in this second client configuration are not the same as those used in the first model.

These two modularizations of the Tic Tac Toe application are meant to represent configurations that might be useful under different conditions. In the first model where the central game server handles the game progress, the client is relatively "thin". In contrast, in the peer-to-peer model, the client is handling all the computation and game logic and is therefore more computationally intensive. We imagine that the peer-to-peer model could be useful if the player were using a desktop and had the computational resources and network bandwidth to handle the game logic and message processing locally. If, on the other hand, the user were playing the game on a mobile phone or a machine with insufficient resources to support the second model, the "thin client" model might be better suited.

The scenario under consideration is the example described earlier. Imagine that a user is playing this game on their cell phone using the first configuration with the "thin" client. If they now walk into a room that has a desktop, they might want to start playing the game on the computer that offers a larger display and more computational resources. Having to completely stop and restart the game would mean that the user has to replay the entire game, something that they would prefer not to do. However, if, the game could be switched from the first configuration to the second configuration without completely losing the game state, the user can keep playing the game without much disruption. In order to make this possible, one needs to be able to checkpoint the game state in such a way that it does not lose its meaning when they switch between the two configurations. Once the game state is saved, the second issue is using that state to continue the game with a different configuration.

Though this thesis focuses on the game as a particular example of a class of applications that could benefit from dynamic rearrangement, the issue being dealt with has a broader scope. Through this prototype and the example scenario, I explore more general issues related to semantic checkpointing and reconfiguration. The next section examines the general idea of checkpointing and dynamic reconfiguration as applied to client-server and/or mobile applications.

2.3 Semantic checkpointing and reconfiguration overview

Checkpointing of the software modules involved is the first step towards enabling adaptive, dynamically reconfigurable applications. During checkpointing and the later reconfiguration, it must be ensured that the application continues to provide the same high-level functionality though the modules providing that functionality might be different. One of the key issues associated with checkpointing in our framework, and addressed in detail later, is being able to capture the state of the system in a form that is meaningful at that high level of abstraction. Inspired by Chan [2], this application framework captures only those variables/conditions that have semantic significance to the application. This is somewhat in contrast to the approach taken by systems that focus on module-level modifications. Such systems focus on capturing the low-level details of the running process, making such a checkpoint dependant on the particular environment. Once the system state is extracted, this snapshot can be used to "initialize" any new modularization of the system instead of starting from scratch. In order to be prepared for an unexpected change in circumstances, an application has to take periodic checkpoints.

While trying to implement a semantic checkpointing framework for this application, I have tried to investigate how the semantically significant parts of the application can be identified and what issues are involved in trying to capture it in a non-implementation specific way. Before a complete and generalized framework for semantic checkpointing can be proposed, it is necessary to clearly identify all of the pertinent issues. Once it is possible to identify and capture the semantically significant parts of the application, any subsequent crash or change in resource availability need not cause the application to completely restart; the checkpointed state can be used to restart the application in a different configuration without a significant loss of state. Since such a restart might lead to a change in configuration, the application should be able to use the saved state to initialize within any new modularization of the system. This scenario highlights the significance of one of the goals mentioned above: the state needs to be captured in a way that is meaningful to the high-level system definition. Such a high-level definition should naturally be decoupled from the actual implementation of the system.

Any framework for capturing state must also deal with the timing and frequency of the state capture. For this reconfiguration process to appear as seamless as possible, the saved states should be complete, consistent and "recent." There is a trade-off involved here. For the application to give the appearance of continuity the checkpoint should be done as frequently as possible. However, taking checkpoints too frequently might interfere with the main function of the application and cause it to slow down. One of the goals of this project is to try and better understand the implications of this trade-off. In addition, the checkpointing controller has to ensure that the state of different modules can be defined such that all module snapshots are consistent with each other and with the high-level system state. Checkpointing of the system will involve checkpointing the individual modules: one of the key issues is to determine how each of the modules can define its state in a way that is consistent with the high level state of the system. Thus, the overarching goal is to use a prototype implementation to highlight the following issues:

- How to identify the semantically significant parts of the application and define the abstract state that needs to be checkpointed.
- How to define specifications for components and capture module-level snapshots that are consistent and can be used to piece together the system-level state.
- How to initialize a new modularization based on the captured state.
- Determine the optimal balance between the frequency and accuracy of checkpoints.
- Identifying and outlining other issues related to semantic checkpointing and reconfiguration.

Chapter 3

Related Work

Dynamic reconfiguration and checkpointing have been areas of active research for a number of years. Moreover, there has been significant effort aimed at trying to make applications more adaptive and reconfigurable. This section includes a brief survey of the key works encountered in the areas of checkpointing algorithms, the checkpointing process and dynamic module replacement and reconfiguration.

3.1 Checkpointing

Checkpointing is one of the broad classes of techniques that have been developed to improve the reliability of distributed systems. The key assumption behind checkpointing is the constant availability of a stable, reliable storage device. Such stable storage can be used to save periodic snapshots of the application or process while it is running. In the event of failure, this saved state, known as a "checkpoint", can be used to rollback to an earlier state and restart the application. This idea of checkpointing and rolling back to the saved state is central to our approach with one key distinguishing factor. As Elnozahy [6] points out, the traditional checkpointing approach tries to capture everything about the state, including the message queues. However, the approach proposed in this thesis relies on isolating the elements that are semantically significant and only capturing those. This allows semantic checkpointing to occur at a higher level of abstraction than the traditional approach.

A second approach to improving reliability combines checkpointing with logging of recent events [6]. During the recovery process following a failure, the application rolls forward from a checkpoint and uses the log to replay the events in their original order thus re-creating the pre-failure state. The advantage of log-based recovery is the ability to generate a state that is more recent than the checkpointed snapshot. While logging is an important idea in recovery procedures, our current approach relies solely on checkpointing for the following reasons. The type of applications under investigation does not require capturing the exact state the application was in before the failure. This hinges on the assumption that users would prefer to minimize the disruption during reconfiguration or recovery. Trying to reconstruct events from a log is likely to add substantially to the reconfiguration/recovery time. Another relative disadvantage of logging stems from the lack of a semantic approach to checkpointing. Since the key idea is not to capture everything related to the application state, but to capture only that required for continuity, a log of all external events is unnecessary. If the external event in question affects the core system state, it should already be part of the semantic state specification and hence will be part of the last checkpoint. If the effects of the event are not included in the semantic state specification in anyway, it is not central to the semantic state and hence, can be ignored without affecting continuity.

In addition to this idea of checkpointing, the current work also relies on the theoretical work by Chandy and Lamport [1]. Their main contribution was developing an algorithm for detecting the global state of the system during a computation by using an approach that involves passing messages between processes to coordinate the timing of snapshots. They note that the dependencies between processes cannot have a total order but they introduced a "happens before" relationship that has a partial order and can be used to take concurrent snapshots of different processes in the system. Chan [3] proposed a modification to the Chandy-Lamport [1] algorithm that involves the check point signal flowing through the distributed system in the stream of flowing messages such that the causal relationships provide a virtual time at which the state can be checkpointed. Both of these ideas are central to taking snapshots of different components in the system that are consistent with each other.

3.2 Dynamic Module replacement

From the implementation perspective, Toby Bloom was one of the first researchers to address the issue of dynamic modification to long-running distributed applications. In her thesis titled "Dynamic Module Replacement in a Distributed Programming System" [4], she identified the need for providing such a change capability in long-running, distributed programs while maintaining long-term or on-line state. As a first step, she identified the conditions under which static support for software modifiability does not suffice. When the execution of a program is continuous rather than broken up into separate, shorter runs and the program has to retain state across requests, static techniques are not sufficient. Her work puts emphasis on being able to maintain correctness and consistency while any updates are performed. Some of the issues she addressed include the viability of replacement in the presence of atomic transactions and the types of constraints that need to be placed on replacement to ensure that client requirements continue to be met.

Bloom's larger goal seems to be that of defining the semantics of dynamic replacement and enumerating the conditions under which it can be performed safely. In an effort to achieve that she developed a model for safe module replacement and provided an analysis of the legality of conditions for replacement. Another one of her contributions is related to integrating this framework into Argus, a transaction-oriented distributed environment. While working on this integration, she identified that there are limits to module modification without language support.

While Bloom's work presents several of the issues that this thesis investigates, it has a different scope. Bloom's work tries to address the issue of module replacement in the context of a particular programming environment, Argus [4]. In pursuit of that goal, Bloom performs an analysis to determine when a particular sub system in Argus can be replaced with another. In a nutshell while her work provides a good framework to start thinking about issues related to dynamic replacement in a distributed environment, this thesis is investigating a more general approach to checkpointing that is independent of the programming environment and the programming languages used. Furthermore, unlike Bloom, the current approach targets applications that require a complete rearrangement and are capable of handling momentary interruption when the switch occurs.

Another person whose work was used as background material is Christine Hofmeister who addresses the problem of "Dynamic Reconfiguration of Distributed Applications" in her doctoral thesis [2]. Hofmeister was one of the pioneers of the idea of programs and modules having a defined state and the need to capture such a state in any sort of checkpointing and reconfiguration framework. She defines dynamic reconfiguration of distributed applications as the "act of changing the configuration of the application as it executes". Hofmeister offers techniques for three general types of changes:

1. *Module Implementation:* In this type of change, the system's overall structure does not change but one of the modules may need to be altered or moved.
2. *Structure:* The system's topology does not change but new modules may be added or removed.
3. *Geometry:* This type of change, which may be useful for load balancing or software fault tolerance, requires a change in the geometry of the application, defined as the mapping of the logical structure onto the distributed architecture.

The work being described builds on top of the POLYLITH [2] platform developed by Hofmeister, which provides a way to separate the configuration of modules from their implementation, and hence, according to Hofmeister, is a viable platform for dynamically reconfigurable applications. Before Hofmeister's work, the traditional approach had been to write machine-specific programs to capture the program state into an abstract low-level format and then translate it into another abstract low-level format. Based on work by Herlihy and Liskov [7], Hofmeister used the idea of transmitting Abstract Data Types (ADTs) in which a particular implementation of an ADT is mapped back and forth from a canonical high-level representation. For reconfiguration, the invocation of such a translation is determined dynamically. Any time a reconfiguration needs to be performed, the module in question waits until it reaches one of the predefined reconfiguration points at which point it can save and package up the state. If the module then needs to be moved or reconfigured, the new module can be initialized with this saved state. Such an initialization also involves copying queued messages from the old module to the new module. Besides messages, this approach can be used to capture and restore the following items: dynamic data in activation record, temporary values in activation record stack, procedure call/return information in AR stack.

While Hofmeister's work introduced a new way of thinking about dynamic module reconfiguration, there are a number of issues and limitations that remain unaddressed. To use her model, the programmer has to manually figure out where the reconfiguration flags should be placed beforehand. Because of this need for explicit pre-set flags, reconfiguration can only be done at fixed points and the program must be designed with this in mind.

While this thesis builds on several ideas developed by Hofmeister, the core idea being presented here is complementary to her work. Unlike Hofmeister's work, the focus of this thesis involves a complete reconfiguration of an application. Instead of replacing or moving one or a few of the modules, I am interested in cases where the application architecture itself completely changes and the application is potentially restarted with a completely different set of modules and interactions. The only requirement is that the abstract functionality of the application should not change between such reconfigurations. For the above reasons, I believe that in the context of Hofmeister's work, the approach discussed in this thesis deals with a fourth type of reconfiguration, one that has not been addressed before.

3.3 Reconfigurable Applications

Inspired by Christine Hofmeister's work, Steven Chan [3] developed a framework for "runtime-reconfigurable applications" that enables programs to change their components dynamically. According to Chan, the framework is not limited to simple substitution of individual components: two completely different sets of components could replace each other as long as both perform the same higher-level task. His proposed framework keeps track of application state while components are replaced and describes how this state can be captured and redeployed while such a reconfiguration is in progress. Chan proposes that in the context of checkpointing, it is not necessary to capture all data values contained in each component. Instead, the focus should be on data that would have semantic significance across reconfigurations. Once the capability to checkpoint in this manner has been developed, regular, consistent checkpoints of all the components in the application can be taken. The task of storing these checkpointed stores can be relegated to a separate checkpoint store. The availability of such checkpoints makes it possible to reconfigure an application in the case of an unexpected failure.

Chan's work is similar to the current work in that it raises the issue of completely reconfiguring an application as opposed to replacing, modifying, or moving one module at a time which Christine Hofmesiter and Toby Bloom's work addresses. Chan also introduces the idea of a semantic checkpoint which keeps track of application state in a manner that is not dependant on language or environment specific elements.

However, a major difference between this thesis and Chan's work is the scope and intent of the framework. Chan intends for his framework to be used in conjunction with a particular planning engine. This general-purpose planning engine can monitor a running application and whenever a reconfiguration is called for, the engine can aid in performing it in an automatic fashion. Chan designed and tested his framework with one such planning and reconstituting engine – pebbles and goals. Pebbles and Goals [15], two projects undertaken at MIT Computer Science Laboratory, provide an environment that enables programmers to write modular, distributed systems whose specification can be disembodied from the actual implementation. Chan points out that in a general-purpose semantic checkpointing framework, each application would be able to specify the state variables it considers significant to be captured during a checkpoint. Given such a specification, the framework would have the ability to create abstract state. This abstract state would be extracted and captured from the individual components and would be a collection of module states. Once it had been translated to an abstract state, this could be used as the basis of reconfiguring by being distributed to a different but functionally similar set of components.

However, as Chan himself points out, his approach does not address all of the above-mentioned issues. It focuses on the mechanics of the actual checkpoint in the context of the Pebbles and Goals environment. While the checkpointing algorithm could potentially be adapted to other execution environments, Chan's thesis leaves this task for future endeavors. This thesis deals with several of the more general concerns that Chan's work did not address. He raised the issue of checkpointing using an abstract state model in the context of a particular execution environment and the current work generalizes this notion and investigates how such an idea could be used to capture application-level state regardless of the execution environment used. Designing a system or an environment that facilitates such a checkpoint would require incorporating these ideas into an execution environment. Such an attempt is outside the scope of this work, which focuses on identifying the issues and problems that form the heart of semantic checkpointing and reconfiguration in a distributed system capable of running in heterogeneous environments.

Along the same lines, Vadhiyar and Dongarra proposed SRS[9], a framework for developing malleable and migratable message-passing parallel applications for distributed systems. This framework includes a user-level checkpointing library and an accompanying runtime support system. Users of parallel applications have to insert calls in this program to specify the data for checkpointing and to restore the application. The supplied library internally performs the actual storing of checkpoints and the redistribution of data. Vadhiyar and Dongarra state that their checkpointing infrastructure allows applications to migrate data across heterogeneous machines even those that do not share common file systems without requiring user to migrate data.

Although this work addresses the issue of checkpointing and porting data across heterogeneous platforms, it does not completely address the issue central to this thesis. The SRS framework stores the checkpoint data as raw bytes, which makes it inherently platform dependant. In addition, it's not suitable for "multi-component applications where different data can be initialized and used at different points in the program". It is most suitable for iterative applications and it requires programmers to insert checkpointing calls at the beginning and end of every loop. In addition, the checkpoint library only supports native data types. For these and other reasons, while SRS is a step in the right direction, it leaves a lot of issues unexplored. It is these issues and more that this thesis takes up.

Magee, Kramer and Sloman's CONIC [5] is another system that tried to enable dynamic reconfiguration of distributed systems. The Conic environment provides a language-based approach to building distributed systems. Besides the Conic language, it provides a set of tools for program compilation, configuration, debugging and execution in a distributed environment. In addition, a separate configuration language is provided to specify the configuration of software components into logical nodes. A logical node is the system configuration unit and consists of a set of tasks executing concurrently within a shared address space. An application is implemented by task modules and definition units using the Conic Programming Language and these tasks can then be partitioned into logical node types. Within this context, compiling a logical node type results in an executable code file. The initial construction and subsequent modification of an application is carried out using a configuration manager which allows the user to create instances of logical nodes at specified locations within his network. These instances are interconnected to form the logical application locations within his network. The set of task and group types from which a node type is constructed is fixed at node compile time but the number of task instances can be specified by parameters at node creation time and can be run on any hardware location (conic handles data type transformations between different processors) and with any logical configuration file. The support for dynamic configuration comes from the capability to dynamically create, interconnect and control these logical nodes.

This approach seems to work for process-oriented applications that need only to move, add, and remove processes such as monitoring and communication systems. However, it is not clear that this framework would be able to perform dynamic reconfiguration in the case of a complete rearrangement of modules such that only the abstract functionality remained the same.

There has been some work on dynamic reconfiguration of distributed systems in the context of object-middleware as well. Almeida, Wegdam et al [10] introduced an approach to dynamic reconfiguration of distributed systems using CORBA standard mechanisms [8][10]. Their approach is inspired by the work done on the Conic environment described above. Object middleware, such as CORBA, facilitates the development of distributed applications by providing distribution transparencies to the application developer. According to the CORBA model, distributed applications consist of a collection of objects which could possibly be geographically distributed. An object is an identifiable, encapsulated entity that provides services to other objects through its interface. According to the model proposed in this paper, a distinct module produces a specification of well-defined changes and constraints to be preserved. These changes are specified in terms of entities; operations on these entities, and a change management module apply these changes to produce a new configuration. The change management functionality needs to guarantee that (i) specified changes are eventually applied to a system, (ii) a correct system is obtained, and (iii), reconfiguration constraints are satisfied. The proposed model satisfies this criterion in CORBA-specific manner by extending the environment. Details of the model would require an in-depth explanation of the workings of CORBA, which is outside the scope of this work. While this approach could work for applications written using the CORBA platform, it by no means solves the general problem of dynamic reconfiguration.

3.4 Conclusion

This chapter provides a brief survey of several seminal projects that are relevant to the current project. There is a large body of work exploring different aspects of checkpointing and reconfiguration but issues related to *semantic* checkpointing and reconfigurations have not been fully explored yet. The subsequent chapters describe our model application and how it was used to implement and investigate issues related to semantic checkpointing.

Chapter 4

Application Design and Architecture

As mentioned in the previous chapter, a model application has been developed as part of this thesis to explore the issues involved in semantic checkpointing and reconfiguration. The idea behind this exercise is to establish general guidelines for developers who wish to make their applications dynamically configurable using a semantic checkpoint approach. Moreover, understanding the various aspects of semantic checkpointing and reconfiguration is the first step towards developing a complete framework for on-the-fly configuration changes. As part of developing a framework capable of supporting a diverse set of applications, incorporating support from the operating system or middleware being used, becomes critical. Following this line of reasoning, this thesis will identify the key features that any such environment can provide to make it easier for application developers to design and develop semantically reconfigurable applications.

4.1 Motivations and design considerations

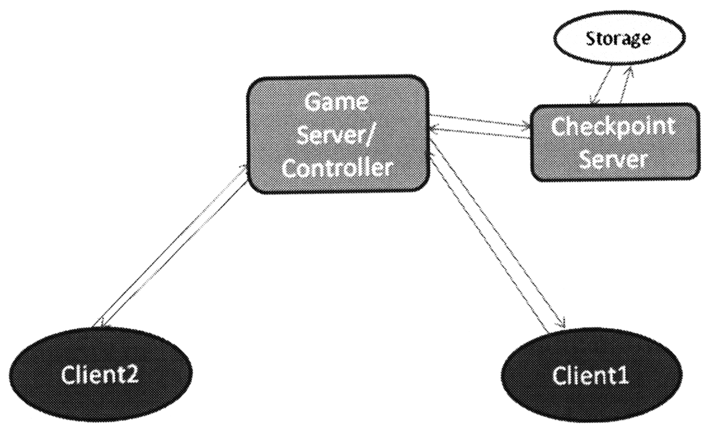
As the prototype application is being used as a tool to explore the key issues involved in dynamic reconfiguration and checkpointing, it must fully capture and highlight the myriad relevant issues. In essence, it needs to be a carefully chosen representative of the class of applications of interest. Client-server architecture is one of the common characteristics of applications meant to run in a distributed environment. Since the client and server can operate in many different environments, with varying degrees of network availability and computational resources for instance, applications with such architecture inherently capture several of the complications related to dynamically changing runtime conditions and raise interesting issues related to checkpointing.

Another consideration regards the complexity of the application. The chosen application must be sufficiently complex in order for it to exhibit the subtle issues associated with adaptive systems. At the same time, it must be simple enough to allow concise description of its architecture and discussion of the various components involved without employing convoluted explanations. With an unnecessarily complex application, we could get bogged down in the implementation details and innumerable interactions of the system missing some of the core issues related to checkpointing and reconfiguration. In order to avoid running into these problems, it was necessary to ensure that the application is simple enough to be explainable without providing an overwhelming level of detail.

After weighing these tradeoffs, I decided to develop an online game that, despite being simple, demonstrates the issues related to functioning in an uncertain or varying environment. The varying conditions could stem from changes in network bandwidth availability and/or changes in hardware and processing power availability. At different times, a user might have access to different types of computers that represent a whole spectrum of communication speeds and hardware and device availability. As explained later, the interaction between the server and the client will vary depending on the nature of the client and the server. For instance, a client functioning in a low-bandwidth environment might choose to minimize the number of messages being sent across the network and hence perform more of the game's required computation locally, whereas a client with limited computational resources would prefer to delegate the majority of the work to the server, bandwidth allowing.

The prototype application developed for this thesis is a two-player game of Tic-Tac-Toe. This particular application was chosen as a simple example that can demonstrate the utility and need for the complete rearrangement of the application components. The details of the component architecture and the different configurations of the game are discussed below. The figure below is an architecture diagram of the first configuration of the application.

Figure1: Architectural diagram of the first configuration of Tic Tac Toe (Central Game Server) *This diagram ignores the interactions in case of multiple controllers. The arrows indicate communication channels.*



4.2 Game configurations

The game can run under two distinct configurations. These are meant to represent disparate types of configurations that are suited for different levels and type of resource availability as discussed in chapter two.

4.2.1 First Configuration

The first configuration is based on the normal client-server model. Each user runs the client application on their personal computer or mobile device and the game is played via a central server. The client-server configuration is made of the following application components, all of which have been developed in the Java programming language. The figure above approximately models the interactions of the different components in this configuration and in the section below the individual components and their interactions in this configuration are explained.

The Client:

In the first configuration, the client is relatively "thin" and consists of a GUI component that responds to user inputs and keeps updating and displaying the state of the game. It also includes a component that discovers and communicates with the game server.

The Game Server:

The game server consists of several components designed to control and coordinate the game. It keeps track of the state of the board and the game, coordinates the turns of the two players and makes sure the game is running according to the rules. The game server's main functionality conforms to a known interface, which it advertises as a JINI service. Before going on to explain the rest of the game architecture, we will make a brief detour to explain the JINI technology. JINI is a java-based technology that defines a programming model to enable construction of secure, distributed systems consisting of network services and clients. Using JINI technology, the game will be offered as a service over the network. JINI provides a lookup protocol that clients can use to look up services based on interface names or service attributes. We decided to use JINI because it takes care of the low-level details related to network communication for service look up and allows us to focus on the high-level system design.

The Controller:

In order to make the configuration scalable and fault tolerant, this arrangement supports multiple game servers. The controller component keeps track of which clients are available and which game servers they are interacting with. The controller also sends checkpointing signals to the different components of the system. To keep the implementation simple, the game server can also serve as the controller if the number of clients and/or game servers is small.

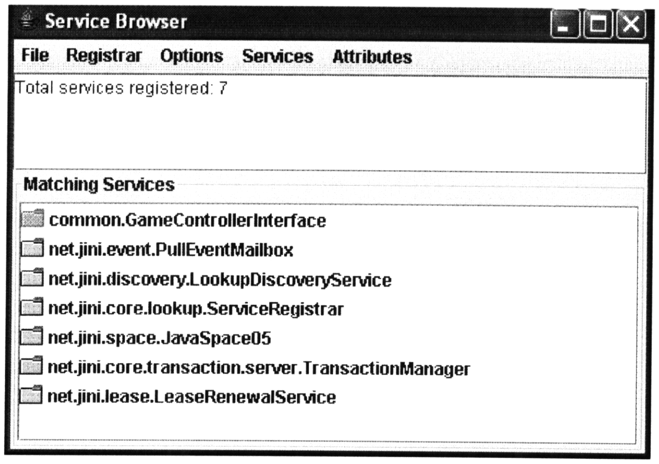
The Checkpointing Server:

The checkpointing server interacts with a checkpointing store and, by doing so, provides functionality to save and retrieve checkpointed state. In this implementation, the checkpointing store was chosen to be a database. In practice, such a checkpointing store forms a central point of failure and hence should contain more redundancy. For the purposes of the demonstration, however, failures in the checkpointing store are ignored.

In the first configuration, the game server directly communicates with the checkpointing server whenever it needs to save or retrieve game state. Checkpointing may occur under two conditions. The first is automatic checkpointing that occurs in response to a signal from the controller and/or the game server that happens periodically during the course of the game. The second is manual checkpointing which occurs in response to a user's request for checkpointing. In the server-centric configuration, since the user does not have access to the checkpointing server, the game server handles the checkpointing signal from the user. The game server acts as an intermediary and completes the checkpoint by transferring data between the client modules and the checkpointing server. As discussed in additional detail later, giving the user some control over the frequency and timing of checkpointing is an important step in allowing dynamically reconfigurable applications.

Any user interested in playing the game, in the first configuration, begins by launching the client GUI application. On starting up, the client application registers with the JINI lookup service. This lookup service enables it to search for a server offering the particular game interface using a unicast or multicast mechanism. Details of this communication are handled by the JINI lookup service but the option of performing a multicast search means that the client does not need to know the physical location of the game server in advance. The game server registers its interface with a JINI lookup service once it is launched enabling it to be discovered by clients interested in that service. The figure below shows a JINI service browser that includes a registered game server for Tic Tac Toe.

Figure 2: JINI browser showing game interface advertised as a JINI service

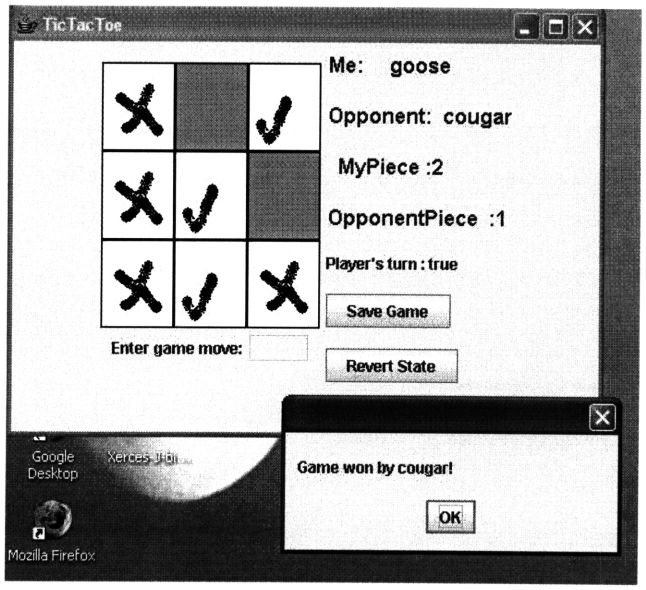


Total services registered: 7

common.GameControllerInterface
net.jini.event.PullEventMailbox
net.jini.discovery.LookupDiscoveryService
net.jini.core.lookup.ServiceRegistrar
net.jini.space.JavaSpace05
net.jini.core.transaction.server.TransactionManager
net.jini.lease.LeaseRenewalService

Once the game server has been discovered, the client sends a message to the server expressing interest in starting a game. The game server waits for these messages from clients. Once it has messages from two clients, it starts a game between those two clients and informs them of their respective opponent. In order to keep matters simple, the game server assigns the first turn to whichever player sent the initial request first. The players are allowed to choose their own pieces but in the event that both choose the same piece, the server breaks the tie based on the client's userID. As soon as the clients get notification of the start of their game, they start polling the server for the game's status. Before making a move, a client ensures that it has the latest game state including the last move made by its opponent. When a particular player's turn comes up, they decide their next move locally and communicate with the application using the client GUI. This move is sent to the central server that then validates the move. If the move is valid, the central server switches the turns of the players and lets the second player make a move. However, if the turn is invalid, perhaps, because the square is already occupied or out of bounds of the game board, the server notifies the user and requests another move. This limited checking mechanism ensures that the game progresses in a fair, seamless manner.

Figure3: UI snapshot from one user's perspective.



Me: goose

Opponent: cougar

MyPiece :2

OpponentPiece :1

Player's turn : true

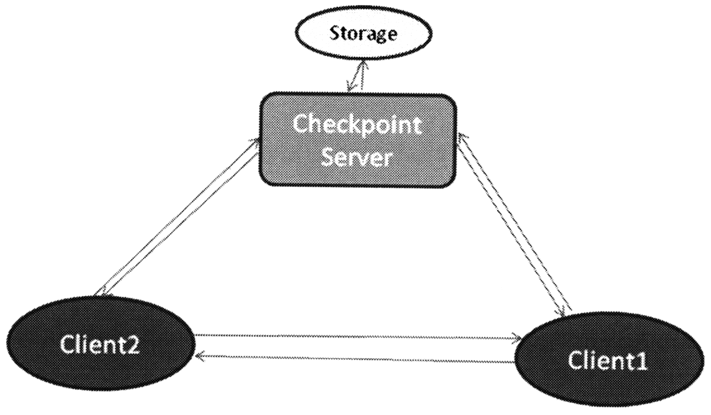
Game won by cougar!

The game server also maintains a communication channel with the checkpoinitng server, which is a JINI service and hence dynamically discoverable. Periodically, the game server captures the state of the game and sends it to the checkpointing server for storage. This checkpointing can occur in response to a signal from a controller (automatic checkpointing) that periodically sends such signals or in response to requests from users (manual checkpointing) if and when they decide to save the game state. Such requests could be the result of a user wanting to reconfigure the game and hence sending explicit requests to save the most recent state or the result of a change in circumstances. The reconfiguration process is similarly coordinated: when the user wants to reconfigure, a signal is sent to the game server, which retrieves the latest state from the checkpointing server. This retrieved state can be used to restore the application in the same or a different configuration.

4.2.2 The second configuration: the peer-to-peer model

In the peer-to-peer configuration, most of the functionality is removed from the server. The central game server's only task is to enable users to find one another. Anyone interested in playing the game sends a message to the server. Once the server has received messages from two players, it notifies each of the players of their opponent's IP address. After this point, the game server has very little to do with the game which progresses directly between the two players. In addition to the limited functionality of the server, this configuration also does not have a controller. Since the central server is no longer responsible for coordinating checkpoints, it is a very simple module and does not need the additional complexity of the controller's functionality. The figure below shows the component architecture for this second model.

Figure 4: Architectural Diagram of the second configuration of Tic Tac Toe (Peer-to-peer model)



The Checkpointing server:

Even though it lacks a controller and a central game server, the second configuration includes a checkpointing server capable of saving and retrieving the game state from the actual checkpoint store. In this configuration, however, the clients access the checkpointing server directly. The checkpoint server provides only provides a communication interface with the checkpoint store. As a result, the clients are responsible for performing consistency checks for the snapshots that are saved and retrieved. This mechanism will be discussed in more detail as part of the checkpointing algorithm.

The Client:

The client modules provide most of the functionality in this configuration. The graphical interface is not very different from the first configuration. In order to play a game, each user sends a message to the central server: the server coordinates the initial rendezvous between the two players. Once the player has its opponent's IP address, it sends a message to the opponent with its userID and the piece it wants to play with. Whoever sends the first message gets the first turn or if it's not possible to agree on the turns using the timestamp, alphabetical ordering of the names is used to break the tie. A similar arbitration can be used to agree on the player's pieces.

Once the initial handshake is complete, the game can begin. When a player's turn comes up, the client application prompts that user for a move. After the user has entered a move, it is validated by the client application. If the move is valid, the move and the new state of the board are sent to the opponent. The client module keeps prompting the user for a valid move if the user's move represents a square that is nonexistent or already occupied. Because each user has information about the updated state of the board and the last move their opponent made, they should be able to choose a valid move. As soon as a user enters a move that finishes the game, either because the board is full or the user has won the game, a message is sent to the other user informing them of the outcome. While added preliminary checks have been added to make ensure that users do not cheat, a complete analysis of the mechanisms involved in ensuring a fair game are not discussed. Any implementation of a game relying on a client-client interaction without a central server would need to analyze and deal with this issue extensively, but as this is not central to checkpointing, a detailed analysis is not contained here.

The two models described above are meant to represent configurations that would be useful under different conditions. In the first model where the central game server is responsible for coordinating and controlling the game, the client is relatively "thin". In contrast, in the second model, the client is handling all the computation and game logic and is therefore more computationally intensive. Such a model would be useful if the user had access to a computer, such as a reasonable desktop, with computational resources to handle the game logic locally. If, on the other hand, the user were playing the game on a mobile phone or a machine with insufficient resources to support the second model, the "thin client" model might be better suited.

In order to analyze the effectiveness of the proposed mechanism, this application was used to simulate events such as user response to change in resources, failures and issues arising due to mobility. The application keeps taking checkpoints of the game state in anticipation of change in resources and when such a change does occur, the controller or the user can initiate a switch between the two different configurations. The next chapter discusses the checkpointing mechanism and reconfiguration events in further detail.

Chapter 5

Semantic Checkpointing and Reconfiguration

The previous chapter described the architecture for the application developed to serve as the prototype for these investigations. This chapter builds on the groundwork laid thus far and demonstrates how the application enables semantic checkpointing and reconfiguration. For semantic checkpointing and reconfiguration to be supported, it is necessary to define a protocol that explains the behaviors and interactions between different components of the application at each stage of checkpointing and reconfiguration. In the context of the application, this chapter discusses in detail the protocol that governs the roles and interactions of the application components to ensure that the application state can be captured at the semantic level, thus enabling it to provide the same high-level functionality even under different application configurations.

As part of the checkpointing and reconfiguration protocol there are several issues that must be addressed pertaining to the timing, initialization and coordination of the different stages in the checkpointing and reconfiguration process. Several of the issues discussed in the following section include: determining what to checkpoint, the timing and frequency of checkpoints, ensuring the consistency and completeness of snapshots, the role of checkpoint storage, and identifying which snapshots to use during reconfiguration.

5.1 Semantic State definition

One of the key issues during checkpointing is determining what constitutes the state that must be saved and retrieved. One possible approach is to capture everything in the environment - all data and state variables and all in-transit messages. The details of the checkpointing algorithm used in the prototype application will be discussed in the next section but one essential aspect must be emphasized here: the application captures the minimum set of variables, conditions and messages that are essential to capturing and restoring the state of the game as defined from a high-level semantic perspective. One of the reasons for choosing this simple application is the resulting simplicity in the definition of semantic state. The high level functionality in a game of Tic-Tac-Toe is, at a minimum, a combination of the following attributes:

- The players involved in the game: every player in the game is assigned a unique ID.
- The mapping between the players and the pieces to which they are assigned.
- The state of the board, i.e., the current position of all the pieces.
- Which player's turn it is.
- The outcome of the game if the game has completed.

Other attributes such as a client's color preferences for the pieces, the type of GUI they were playing with, the physical location of the GUI relative to the screen, are not captured. The idea behind capturing semantic state is not to save every single variable in the system but to determine the minimal set of parameters which fully describe the system state such that the high-level functionality can be maintained. An important aspect of this state definition is its independence from any particular implementation and configuration of the game. Since the proposed approach defines the state in terms of a high-level picture of the game, rearranging the modules, changing the implementation, or moving the game to a machine with a different architecture does not affect this definition. Such a decoupling of state definition from low-level system issues allows the state of the game to be captured in one configuration and restored in a different configuration.

Deciding how to capture the state in a manner that preserves these properties is one of the key issues involved in enabling dynamic reconfiguration. In order to have a complete and consistent state of the application, the contribution of different application modules to the global state needs to be predetermined. There are two stages to ensuring that the snapshots captured by each module can be combined to produce a global state that fully captures the high-level functionality of the game. In the first stage, we must define as part of the checkpointing protocol what constitutes a complete and consistent snapshot of the application, an issue which is discussed in this section. The second issue discussed in the next section is ensuring that the snapshots that are ultimately taken by the components adhere to the agreed upon specifications.

With regards to developing the specification of a complete application checkpoint, one plausible approach is to incorporate this reasoning process in the design phase. The application developers can agree on the key attributes that should be checkpointed and develop a state definition based on these attributes. Before delving into the issue of state definition further, another important point must be raised. In order for programmers and/or modules to agree on the key set of attributes, the application must have well-defined interfaces between modules. Besides the usual benefits derived from having a modular design with clear interfaces, such a design also makes it easier to narrow down the key attributes that make up the abstract state.

While the key idea behind developing a complete checkpoint specification applies to all types of applications, the details and complexity of the process can vary substantially depending on the type of the application. In the case of applications with a simple design and few interactions, developing a state definition as part of the design process may not be a complicated affair. However, in applications with many modules and a large number of interactions, this process will require careful analysis and a considerable number of iterations. Even after extensive analysis and reasoning, a state definition developed during the design phase might not suffice for all types of applications. While the use of a static state definition simplifies the issue and works for small applications, it does not scale well for more complex and rapidly evolving applications. A more adaptive and scalable approach would be to agree on a set of key attributes as part of the design process and use these features as a starting point for the abstract state definition. This initial definition should, at least for more complex applications, be refined using user feedback. The degree and manner of user feedback is likely to vary a lot with the type of application involved. For instance, an application that organizes and plays music might ask the user which granularity they prefer for this semantic state definition- keeping track of the last song being played and being able to replay that song if the application crashes and then recovers or a finer granularity which keeps track of and recovers the exact note in the song that was being played before the crash. In the game example, besides the core attributes that define the game, it is possible to give the users the option to specify additional attributes they would like to save. A few examples of these attributes are the type and size of the client GUI, the colors and the shapes used for the pieces and the manner and degree of notifications received.

Incorporating real-time user feedback requires an adaptive design, which would naturally lead to additional complexity. Such a trade-off between an adaptive state-definition and static but less complex definition is one of the many trade-offs encountered when designing an application with built-in checkpointing functionality. The particular balance of manual and automatic checkpointing chosen depends on the type of application as well as user preferences.

5.2 Checkpoint Storage

An additional concern with regards to checkpointing is the issue of storage. One possibility, and one that is used in the current approach, is to save the application checkpoints in a centralized checkpoint store. Another possibility is to store snapshots from different modules in separate checkpoint stores either externally or within the modules. It can be argued that choosing a single checkpoint store leads to a central point of failure and hence should be avoided if a distributed checkpointed option is available. However, there are a few caveats to this line of reasoning. While saving module snapshots separately offers increased reliability, it comes at a cost. The separate snapshots need to be aggregated at the time of recovery or reconfiguration; the coordination required to piece together the application snapshot could slow down the reconfiguration process. Such a move could also lead to additional design constrains increasing the complexity of the application design. Given this trade-off, a centralized checkpoint is used in the Tic Tac Toe game application. A centralized checkpoint store simplifies the recovery procedure. Furthermore, the central checkpoint is an external service and can use several replicas to offer increased reliability. Another alternative is a Distributed Hash Table (DHT)-based storage system that provides distributed storage in a single unified id space. This may have the advantage of allowing for parallel storage of individual components and the resiliency inherent to modern DHT's, however it would require distributed recovery as well.

5.3 State Representation and Data Type Transformations

In addition to the definition of application state, the checkpointing framework must specify how the global state will be represented. The actual variables being used in checkpointing may not necessarily have a one-to-one correspondence with the system variables found in the different components of the application. The checkpoint variables could be a combination of variables found across different modules. The mapping or transformation between the checkpoint variables and the actual program variables is something that must be determined before a checkpoint mechanism can be completed.

A transformation such as this brings up two distinct issues related to abstraction and representation. The last section discussed the set of variables and parameters the application developers and/or users specify to be included in the game state definition. These variables and conditions form the abstract state definition but for the checkpointing server to store this state, this abstract definition needs to be translated into a representation.

In practice, the checkpointing server and the other parts of the application are likely to be designed by different programmers. Following common practice, they would agree on an interface but not necessarily the implementation details of the different components. From the point of view of checkpointing, this means that the checkpointing server implementer would have access to the abstract state definition but not the representation used by the application modules for the abstract state. This essentially leads to a situation where the checkpointing server's representation uses different data types than the ones used by the core modules of the application. Given this disparity, the checkpointing server, or some other manager in the middle, needs to be able to transform the data types used by the application to those used by the checkpointing server. This reasoning is based on the assumption that the abstract representation used by the checkpointing server and the application is the same but in practice even that need not be the case. The checkpointing server and the application only need to agree on the representation they use to communicate the state definition. The internal representation used by the checkpointing server could be different from this external definition. The existence of these different representations necessitates additional transformations.

Despite the simple state definition of the prototype Tic Tac Toe system, it requires data transformations at several stages. The different configurations of the game and the checkpointing server use different representations for the game board and the players. The game board is represented as a multidimensional array of integers by the game server but has an xml-like string representation when stored by the check pointing server. The player pieces are represented as integers as part of the game server but as strings in the checkpointing server. Similarly, other attributes such as player IDs are represented differently at different positions in the system. Each transformation required at a given module interface is accomplished using a transform function.

The need for such transformations requires a mechanism that facilitates communication between modules: the different modules of an application must communicate their state amongst themselves or to a central controller. In order for communication between modules to work seamlessly it is necessary to specify a definition language. One idea worth clarifying is the distinction between the definition language and the implementation of the transform function mentioned above. The transform function is needed to translate from the system/program variables to the checkpoint variables that represent the abstract state whereas the definition language being used governs how the checkpoint variables are expressed.

With regards to choosing a definition language, one possible approach is for each module to have its own syntax and definition language. Such an approach, however, leads to increased complexity. With a combination of languages being used, there arises a need for a manager/coordinator that can translate between different definitions. Consider an alternative approach: designers prescribe a common definition language and enforce its usage in all modules. Besides simplifying the design of the application, the latter approach would reduce the overhead associated with communication amongst different modules and the central controller. If a common language is being used, the designers need to carefully analyze such a choice. The common definition language should have a broad enough repertoire to capture the parameters of all the modules concerned. At the same time, it must be simple enough that communication between various modules is possible without encountering enormous overhead. For very specialized applications, the application developers might also consider developing their own definition language designed to capture the key aspects of the particular application. Taking this option into account introduces a further trade-off. Developing a new language offers the possibility of greater flexibility and ease of use but at the cost of additional man-hours. An existing language removes the need to expend additional effort on development, but may also introduce some limitations because of rigidity in existing syntax.

5.4 Consistency and Completeness of Application State

A global checkpoint is a combination of the various snapshots collected from different components of the application. For the checkpointed application state to be correct, the snapshots from each of the components need to be consistent with one another. In configurations that involve a central server with access to snapshots from all components, the central server can ensure that the checkpointed snapshot formed as a result of the component snapshots does not violate the invariant established for correctness. In configurations that do not have such an entity, ensuring correctness is harder.

In order to see how an application snapshot can be incorrect, imagine the following scenario in the peer-to-peer configuration of the Tic-Tac-Toe application. Player 1 makes a move, takes a snapshot and subsequently sends the move to Player 2. In the snapshot taken by Player 1, their last move is included and the next turn belongs to the Player 2. However, Player 2 fails to receive this message and takes a snapshot before Player 1's move is included. In the snapshot taken by Player 2, the next turn belongs to Player 1. Simply combining the individual states from these two player components would result in an inconsistent state. Both players think it is the other player's turn. This example is somewhat contrived but it can be easily seen that in applications with more complicated interactions, missed messages can easily lead to modules capturing state that cannot be used to form a consistent application state. This is very much analogous to conflicts encountered in concurrent versioning systems (CVS) used by large teams of software developers. In some checkpointing frameworks, logging has been used to work around this problem. The application logs recent messages as part of the checkpoint. The idea is to resend the messages during recovery and recreate the state before the reconfiguration. Semantic checkpointing, however, cannot take advantage of this strategy. Since the types of reconfigurations we are dealing with, include complete rearrangement of all the application modules, the modules and their interactions after a reconfiguration may be completely different from those before the reconfiguration. As a result, the messages saved before a reconfiguration are likely to be meaningless in the new state. Hence, instead of using logged messages semantic checkpointing relies solely on ensuring consistency at the global checkpointing level before storing each checkpoint.

5.5 Checkpointing mechanism

The application checkpoint is a combination and aggregation of snapshots taken by individual modules. One approach to creating this global snapshot is for individual modules to independently checkpoint their local state periodically. These local snapshots could then be pieced together to create a global checkpoint. One advantage of this approach is the resulting simplicity: if all modules checkpoint independently, there is no need for a complicated coordination mechanism amongst modules to coordinate checkpoints. This simplicity, however, comes at a price. There is no guarantee that snapshots taken independently by individual modules can be used to compose a consistent application snapshot. Piecing together this snapshot will also require more complicated mechanisms and may slow down the reconfiguration phase. Using independent snapshots to form a global snapshot becomes more challenging as the frequency of messages passed between modules increases. Even if these module snapshots are synchronized using a third-party or a synchronized clock, there is no guarantee that the module snapshots will be consistent. State changes caused by in-transit messages might be captured by one module and not by others leading to inconsistencies.

An alternative approach is to coordinate the snapshots amongst individual modules as recommended by Chandy and Lamport's distributed algorithm [1]. This algorithm enables applications to take snapshots without stopping the regular computation. Unlike the independent snapshot approach, this approach also ensures that none of the individual snapshots will have to be thrown away because of inconsistencies arising from missed messages. This process uses marker messages along channels that are assumed to be queued (FIFO). The process initiating a checkpoint saves its own state and sends a marker via its outgoing channels. All other processes, on receiving this message for the first time, save their local state and send it to the process that initiated the checkpoint. Subsequent messages along the output channels include the checkpoint token. The FIFO nature of the channel ensures that the snapshot taken by the downstream process must be concurrent with the snapshot taken by the process from which the marker originated. By inductive logic, in a network where each node only has one input channel, the snapshots are concurrent with the first marker giving a consistent application snapshot.

However, as Chan points out [3], this approach does not work as well when nodes have more than one incoming channel. According to the Chandy and Lamport algorithm, in such a scenario, each channel logs all messages that it receives after the snapshot marker was received. These messages are logged as part of the checkpoint. However, such an approach cannot be taken within our proposed framework. As mentioned earlier, in a complete reconfiguration scenario, the module arrangement after a reconfiguration might be completely different from the arrangement earlier rendering these saved messages meaningless. In order to deal with this issue, the algorithm used in the prototype is a variant of the Chandy and Lamport algorithm that only includes messages that have been completely processed at the time of the checkpoint. While a checkpoint is being taken, further message execution is temporarily suspended to ensure that a consistent checkpoint can be taken.

5.5.1 Timing and frequency of Checkpoints

Reconfigurations fall under two main categories: planned and unplanned. Planned reconfigurations occur when the user anticipates a change in circumstances or the system detects a change in resources such as performance degradation in the current configuration or new resources becoming available. An example is a mobile user moving to a room with a better display or an increase in the available bandwidth. Unplanned reconfigurations occur when there is a failure or some resource becomes unavailable unexpectedly. It should be noted that the correctness and consistency of the checkpoints is a prerequisite for the reconfiguration process to be beneficial in either of the scenarios. Starting the application from scratch would be preferable to resuming in an inconsistent state. With regards to the timing of the snapshots, however, the two types of reconfigurations put different constraints on the checkpointing mechanism.

In the case of planned reconfigurations, a checkpoint is taken immediately before the reconfiguration. Since the application is functional before the reconfiguration, the user will not be tolerant of noticeable disruptions: the checkpointing and reconfiguration have to occur with minimal disruption to the normal functionality of the application. In the case of a sudden unavailability of resources, the user requirements are less stringent. In the absence of the checkpointing capability, if a failure occurs, the application will lose all state and will have to start over. In this scenario, the semantic checkpoint framework needs to ensure that the cost of adding the checkpointing and reconfiguration capability is outweighed by the time and effort required for starting over. Once a complete and consistent snapshot is available, its usefulness depends on how recent it is. In the ideal case, the application would have a checkpoint that captures the exact state before the failure occurred. From the perspective of providing the users with a feeling of continuity even under failures, the application should be checkpointed constantly. The process of checkpointing, however, may cause some disruption to the normal functioning of the application. The benefit of taking checkpoints needs to be balanced against the cost associated with performance deterioration experienced by the user because of the checkpointing activity. Since the negative impact of checkpointing depends, to some extent, on the type of application and the manner of its usage, the application user should be given some control over the frequency and impact of checkpointing. Although a minimum checkpointing frequency can be required by the application, the user should have the freedom to raise the frequency to support a corresponding desired level of continuity.

5.6 Semantic Checkpointing and reconfiguration in the prototype

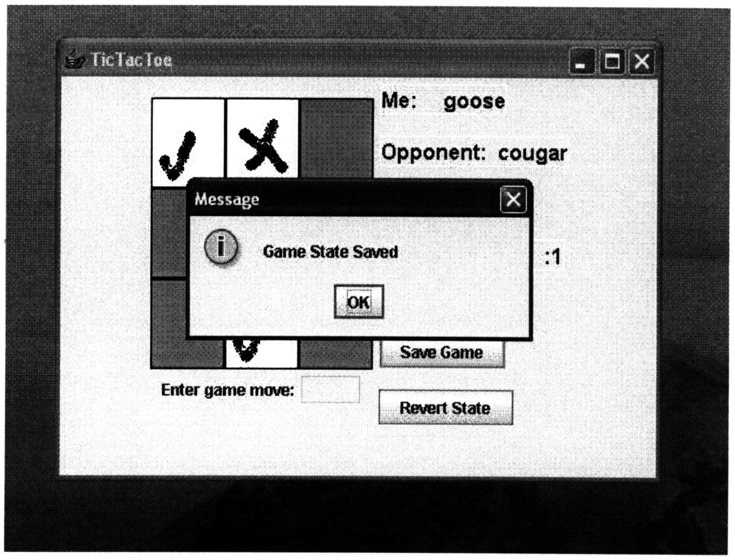
The first section of this chapter discussed the idea of establishing the minimum set of attributes needed for a complete abstract state definition. In the section that follows the checkpointing protocol is described for different configurations of the prototype application; this specific example is used to highlight additional tradeoffs and complexities involved in the checkpointing and subsequent reconfiguration process.

5.6.1 Client-Server Configuration

The prototype application developed for this thesis is relatively simple allowing the core of the state definition to be developed as part of the design process. An XML-like language definition is used to represent the application state. There is no restriction on the internal representation used by the application components but at points of communication between different modules, the messages exchanged follow a pre-specified format. As explained in chapter four, two configurations of the multi-player game were developed. The first configuration is a client-server model. Since there is a central server in this model, the task of initiating and collecting a checkpoint can be delegated to that server. The central server can request the snapshot information from all the modules and convert it to a consistent, canonical representation before passing it to the checkpointing server.

A checkpoint in the prototype application can be initiated as a result of a user request or prompting by the controller. The two types of checkpointing are meant to deal with different types of reconfiguration events described earlier. Periodically, the controller sends checkpointing signals to ensure that the checkpointing server always has a relatively recent state of the game. With such an approach, even in case of an unplanned reconfiguration, the game can revert to a checkpointed state instead of having to start over. How updated this state information is depends on the timing and frequency with which checkpoints are taken, as discussed above. For planned reconfigurations, there is no need to settle for an outdated state; a checkpoint can be taken right before the user or the system initiates a reconfiguration. Since not all reconfigurations are planned, the controller takes periodic checkpoints to ensure that there is a snapshot available in case of sudden failures or changes. At the beginning of the application, the user is given the option to decide how frequently these systems-driven checkpoints are taken. If the application is configured to not checkpoint too frequently, there is a chance that the application state could be slightly outdated. However, as long as the checkpointed snapshots are preferred to starting over, the cost of adding the checkpointing framework is justified. The figure below shows a snapshot of the application GUI after a checkpoint has been initiated; the user is notified once the process is completed.

Figure5: Checkpoint completed confirmation. The user is informed when a checkpoint is successfully completed.



Regardless of who initiates the checkpoint, the central server propagates the initial checkpoint message to other modules. The checkpoint mechanism is based on our variant of the Chandy and Lamport snapshot algorithm [1] discussed above. In the server-centric version, the game computation is being performed on the server side. The game server is composed of several modules that include: the game controller which interacts with the clients and the checkpoint server, the game component which is a container for the various parts of the game, board component which keeps track of the state of the board for each particular game, and a player component that keeps track of the players involved in each of the games. Once the game server starts the checkpointing process, it ensures the checkpointing process is completed before processing further moves. This prevents the inclusion of any in-transit messages in the checkpoint. Once the game module receives the checkpoint marker, it contacts both the board module and the player module. The board module captures the state of the board- which positions the pieces occupy and whether or not the board is full. Similarly, each of the player objects keep track of the pieces belonging to that player and whether or not it is that player's turn.

All these modules communicate their state to the central server using known messages. The format of these messages is developed as part of the design process and is meant to represent the state of each of the modules in a format that can be understood by the central server and the other modules. Once the server has the requisite messages, it performs rudimentary checks to make sure that the information from the different modules can be aggregated to create a consistent game state. In this particular implementation, the server checks for the following invariants:

-Only one of the players thinks the next turn belongs to them,

- Both players have unique IDs and game pieces

-The game board only has pieces at valid positions.

Since the checkpoint protocol uses a variant of the Chandy and Lamport algorithm, the resulting checkpoint forms a consistent picture as long as modules adhere to their checkpoint specification. In the event that one or more of the modules fail to capture the state properly and the resulting global state is inconsistent, the application must have a defined (deterministic) course of action. This scenario also becomes possible if the checkpointing protocol relies on independent snapshots. Once the global checkpoint fails the completeness and consistency test, one possibility is for the controller to inform all the modules of the inconsistency and attempt to take another snapshot. In another approach the controller ignores this snapshot completely and takes another at the next scheduled time slot. The latter strategy increases the relative risk of lost data due to the increased time between valid checkpoints, but requires a much simpler implementation.

Once the controller has a complete and consistent application state, it contacts the checkpoint server which saves the checkpointed snapshot. As mentioned earlier, in both configurations, the checkpoint server uses a centralized database to store the checkpoints. While the checkpoint store is independent of the game server, it could act as a central point of failure. The primary reason for choosing a central checkpoint store, and why it was chosen for the prototype application, is the resulting simplicity of the reconfiguration process. During a reconfiguration, the new components need only be initialized based on the saved state before the reconfiguration. Having one point of contact, simplifies this process tremendously. In order to guard against a centralized point of failure, real-world applications can use multiple replicas of the checkpoint server or replace the centralized checkpoint store with a distributed version. While distributing the state amongst various components and servers does increase fault-tolerance through redundancy, it also increases the complexity of the reconfiguration process and possibly the checkpointing process. As mentioned earlier, DHT's provide one additional option. Deciding between these options is a design decision and depends on the type of application and user preferences. In general, the central checkpoint store will provide the more efficient solution in the majority of cases; only in situations that are maximally intolerant to any loss of data should one consider decentralizing the checkpoint store.

5.6.2 Peer-to-peer configuration

The second configuration of the game lacks a central server relying instead on a peer-to-peer interaction between clients to control and coordinate the game. Since the client cannot rely on the controller to coordinate the game and enforce consistency checks, the checkpointing protocol for this configuration cannot rely on a centralized structure. One key difference in the checkpointing protocol for this configuration is the role of the checkpoint initiator. Given the absence of a hierarchical structure, either of the clients can partially take over the role of the central server and initiate the checkpoint periodically. This can be generalized to the case of arbitrarily many clients, wherein any individual client can assume the role of the central server. If all the clients have an equivalent structure, this role can be randomly assigned to any of the participants or rotated amongst all or some of the clients using a round-robin strategy. In a setting without equivalent client structure, the role of the central server could be assigned to the least resource/bandwidth limited client.

Game coordination and checkpointing involve the exchange of several messages between players in the second configuration. The lack of a central server means that the communication layer of each of the clients must deal with the details of the message exchange. Given the significance of the message exchange, the next section is devoted to explaining the details involved. Since the messages depend on a reliable delivery mechanism, the prototype application uses TCP as the network layer protocol. In practice, TCP might not be well-suited to all applications since the handshake involved in initiating a TCP session involves substantial overhead. For applications that cannot rely on a protocol such as TCP, assurances about message delivery must be provided by higher layer protocols.

By using TCP, the communication protocol used by the application has been simplified substantially. The following messages are used to coordinate the game and perform a checkpoint: CheckPoint, GameOver, Join,Started, Move, RevertResp, RevertState. The "Join" and "Started" messages are used at the beginning of the game to perform the initial handshake between two players interested in starting a game. Anyone interested in playing the game sends a message to a game server. The server's role is limited to keeping track of the players who have expressed interest in the game but have not yet found someone to play against. Once there are enough players to start a game, which in this case is two, the server notifies the players of their opponents. The players can then start the game by sending messages to their opponents directly. Once the game has started, the players use the "Move" and "Gameover" message to exchange game moves and to indicate that the game has finished. In order to initiate a checkpoint, a "Checkpoint" message is used. The last two messages are used during a reconfiguration. When one of the clients wants to checkpoint the game, it takes a snapshot based on its view of the game. Since the players exchange information after every move, a snapshot taken by either player will include all the moves that have been fully processed. This mechanism ensures that as long as players adhere to the specified protocol, both players have a complete view of the game.

Once the player initiating the checkpoint has taken a local snapshot, it sends a "Checkpoint" message to the other player informing them that a checkpoint has been taken. The message also includes the contents of the checkpoint. As there is no central server, the responsibility for performing consistency checks falls to the two players. When the second player receives the checkpoint message, they check to make sure that the snapshot taken by the first player is consistent with their view of the game. If the second player considers the snapshot taken by the first player to be invalid, they can mark it as inconsistent. Any checkpoint that fails the consistency check will be rejected by the checkpoint server. In this particular configuration, the checkpoint server expects both clients to have approved the checkpoint before it is saved as a valid snapshot.

The checkpoint message from the first player can serve as a signal to the second player to trigger a checkpoint as well. In this particular case, a checkpoint taken by the second player in response to the checkpoint by the first player does not include substantial new information since the checkpoint only captures application level details and ignores any user-specific details. Furthermore, since both players have a complete view of the game from an application perspective, the snapshot taken by one player can be used by both players. However, if the client modules in such an arrangement were independent or substantially different, the module-level checkpoints would have to be captured separately. Having a single module initiate the checkpoint allows the usage of a variant of the Chandy and Lamport algorithm. Just as in the first configuration, this approach ensures the consistency of the saved state.

5.6.3 Identifying snapshots during reconfiguration

Once the user or the system initiates a reconfiguration, the first step is choosing the snapshot that should be used for the reconfiguration. In most cases, this should be the latest consistent checkpoint but we can imagine scenarios in which the user would prefer not to use the most recent one. For instance, if a movie playing application crashes, the user may prefer to rewind to the beginning of the scene instead of resuming from the middle. Given this possibility of choosing from one of several checkpoints, the checkpoint server and the controller need some way to identify the different snapshots so the correct one can be retrieved when needed. As before, this is analogous to the use of CVS in software development: the presence of previous checkpoints adds the flexibility to revert to any previous checkpoint.

Since a particular checkpointing server could be responsible for checkpointing multiple applications, the identifier needs to be globally unique across all checkpoints and all applications. Such a checkpoint identifier would need to be composed of two parts- one identifying the application and another identifying the particular checkpoint instance within the application checkpoints. Additionally, there could be multiple instances of the same application in which case the identifier needs to distinguish between different versions and instances of the application as well. The application specific identifier could also be used to choose the correct transformations. As mentioned earlier, the checkpointing server must perform representation and data type transformation as part of saving the abstract representation of the application state. Since different applications require different transformation types, the checkpointing server is responsible for storing a mapping between the applications and their respective transform functions. The application specific identifier can be used to store this mapping.

Tagging checkpoints is a relatively simple affair in the prototype application. Since the checkpointing server is only required to support one version of a single application, we can omit one part of the global identifier associated with the application, simplifying the identifier scheme. Having a single application also simplifies the task of storing the transform function mapping since the application specific functions can be used as the default. The scheme still requires the assignment of unique identifiers to each checkpoint instance; to accomplish this, the prototype application uses timestamps to distinguish between different snapshots. Since the checkpointing server is the only module adding and comparing the timestamps, issues related to synchronization problems between clocks on different computers can be ignored. During the checkpointing process, all module snapshots use the timestamp included in the initial checkpointing marker. When the user wants to revert to the latest saved state, the game controller requests the latest snapshot from the checkpointing server. Upon receiving such a request, the checkpointing server searches for and retrieves the snapshot with the latest timestamp. The same mechanism can be used to retrieve any other checkpoint. Once retrieved, the saved snapshot can be used as a starting point for the reconfiguration process.

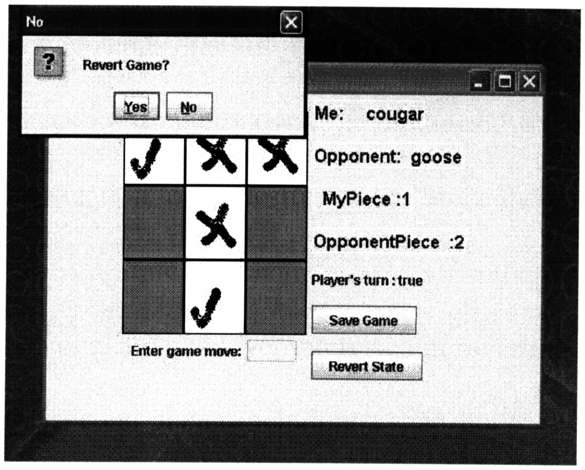
5.6.4 Recovery and Reconfiguration

The snapshot taken by the players in the peer-to-peer arrangement includes the same information as in the client-server configuration. This includes the state of the game board, the identifiers for the players involved, their respective pieces, and the information about the turns of the players. This idea becomes important when there is a need for reconfiguration from the server-centric scenario to the clients-only scenario: the checkpoints are defined in terms of the same high-level parameters making a transition possible.

In both configurations of the game, users have the option to use a checkpointed snapshot to revert to an earlier point in time. This functionality is useful if the system gets into an inconsistent state because of failure of some or all of the modules. Instead of starting over, the users can revert to an earlier checkpointed state and resume the game from that point. In the first configuration, either the user or the system can initiate such a transition. If a user wants to revert to an earlier game, it sends a message to the central server requesting such a move. If the other parties involved do not object to such a move, the central server initiates a "Revert State" operation. After obtaining the desired checkpoint from the checkpointing server, the central server sends this information to the individual modules in a manner that is similar to the checkpointing process. The modules use the saved state to reinitialize themselves returning the application to the desired state. Since there is no central server in the second module, the "Revert State" mechanism works using the same principles that were used in the checkpointing protocol. The player interested in reverting to a previous state sends a "RevertState" request to the second player including a snapshot of the checkpoint they intend to use for the reconfiguration. The second client can reject or accept this request. If they choose to accept such a request, the second client can use the information given in the checkpoint to reinitialize the state of their application. Upon receiving a RevertResp" message from the second client, the first client goes through a similar re-initialization, returning the application to the previous state. The game then proceeds normally from that point onward.

Figure 6: The snapshot below shows the user initiating a "Revert State" operation

The "Revert State" option allows the user to revert the game to an earlier checkpointed state.



Me: cougar

Opponent: goose

MyPiece :1

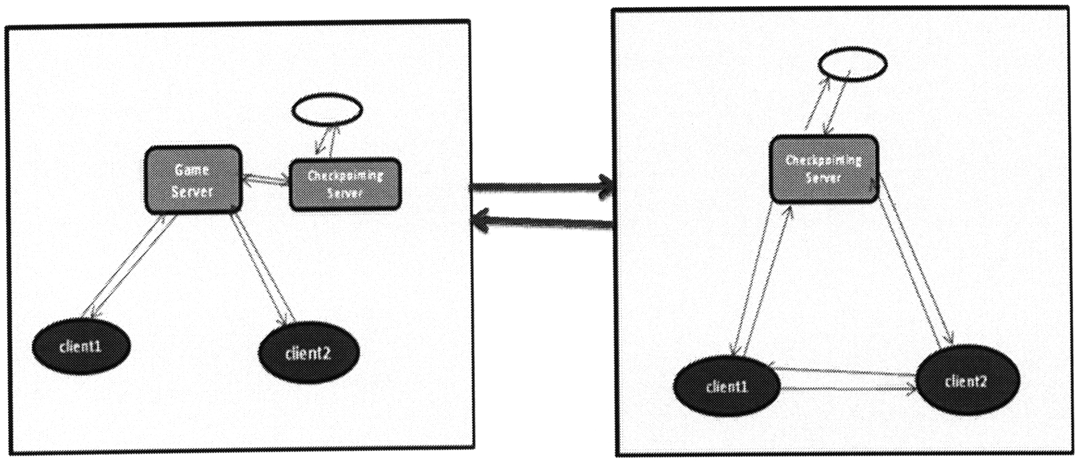
OpponentPiece :2

Player's turn : true

This ability to reinitialize from a saved state is one of the key mechanisms used during a reconfiguration from one application arrangement to a different one. In this particular example, both configurations of the application capture high-level attributes of the game. In the event that a reconfiguration is needed, the application takes a checkpoint and pauses momentarily. The second configuration of the game retrieves the checkpointed state and initializes itself based on the saved snapshot. Once the initialization is complete, the user is informed that the application structure was reconfigured and they may continue the game. The user interface for both configurations is very similar so even though the application structure underneath changes completely, the user can continue playing the game in the same manner.

With the aid of this mechanism, the prototype simulates scenarios in which a failure or change in availability of resources causes the game to switch from one configuration to another. The user can start playing the game with the central server or client-only configuration and rely on the checkpointing mechanism built into the game to take periodic snapshots of the game state. The user can also trigger checkpoints at any point in the game. In the experiment we carried out to analyze this approach, both users start playing the game on laptops connected to a wired network. Given the availability of bandwidth and computational resources, the game starts in the "server-less" mode- the two player clients send messages to one another directly to coordinate the game. Half-way through the game, a failure is simulated as follows- one of the players leaves the room and moves to a location where no wired networks are available. The player now only has access to a wireless network that has a high drop rate. Given the change in the resources, the player needs to reconfigure the game to use the server-centric configuration. The server-centric configuration requires fewer message exchanges and hence necessitates less bandwidth availability. Since the application has the ability to initialize the modules from a given snapshot – the checkpoint taken right before the resource change is used to resume the game in this new configuration.

Figure 7: Modularization change during switch from server-centric architecture to client-centric architecture



In these experiments, the reconfiguration was initiated by the user noticing the change in resources. Ideally, such a transition would be automatic. The application or the environment responsible for providing the reconfiguration capability should automatically detect failures or a change in resources. Based on the type of change, it would automatically choose a suitable configuration and trigger the process required to initiate the reconfiguration. While the automatic detection of change in resources is left for future projects, the current project lays out the groundwork for creating such a framework. As this chapter has drawn attention to the system design issues and lessons learned in implementing the model application, the next chapter will provide future directions extending from this work.

Chapter 6

Conclusion

One of the goals of this thesis is to provide a comprehensive survey of the issues related to semantic checkpointing and reconfiguration. In an effort to understand and explore these issues and tradeoffs, a model application was developed that represents and addresses the issues developers would come across while incorporating semantic checkpointing and reconfiguration capabilities into a distributed system. By developing a model application capable of supporting dynamic reconfiguration with the aid of semantic checkpoints, I have demonstrated one approach to enabling such functionality. Our working prototype demonstrates the possibility and mechanisms for providing such support at the application level.

Chapter 5 surveyed the issues encountered and the solutions employed while implementing the checkpointing framework in the Tic Tac Toe application. This chapter summarizes these findings and extends earlier observations to suggest a set of mechanisms and ideas that would be needed to support a generic checkpointing capability. The design decisions made and the issues we encountered while developing an application capable of supporting semantic checkpointing and reconfiguration can be used as guidelines for developers to follow while designing systems with similar capabilities.

While implementing an application level framework for semantic checkpointing, one of the first issues that arises is the definition of abstract state. Defining the abstract state is a very application-specific task that, at least in part, must be carried out by programmers during the design and development phase of the application itself. The developers must identify the key attributes of the application that are necessary to identify the application's functionality from a semantic perspective. As the attributes might be usage specific, there is a need for a mechanism to enable users to modify and/or make additions to the abstract state definition.

Once these key attributes have been selected, the next step is choosing a representation for the abstract state definition. As part of choosing a representation, a key decision that needs to be made is the representation language. The capabilities of the representation language govern the representation being used for the abstract state. Since these decisions could affect implementation level details, the developers of the application should make these decisions during the design phase of the application.

In almost all real-world applications, the information that constitutes the abstract state would come from many different modules of the application. In addition, the variables being used in the abstract state need not have a one-to-one correspondence with the variables found in application modules. This raises a number of issues. First, the checkpointing server, or whichever module is coordinating the checkpoint, needs to be able to collect the information from many different modules and combine it in a consistent manner to create a meaningful representation of the application state. Second, in order to facilitate this process, the checkpointing server must not only understand the abstract state definition, it must also understand what constitutes the response each module gives in response to a checkpointing request. As a first step, this requirement further reinforces the need for well-defined specifications for each of the modules.

Based on the final checkpointing specification, the checkpointing server or some similar coordinator must generate module-level specifications for the individual modules of the application. There are a number of requirements that these specifications are required to abide by. Firstly, the specification should be implementation independent allowing the checkpointing server to gather data from heterogeneous parts of the application. Secondly, the individual specifications should be complete enough that the generated module-level state definitions can be combined to create a complete and meaningful view of the application state. Once defined, these specifications must be translated into actual implemented objects to be stored by the checkpointing server. The key issue that arises while reconciling different implementations is the issue of data transformation. The representation being used by the checkpointing server would most likely be different from the internal representation being used by the application modules. In order to deal with this discrepancy, there is a need for transformation functions in place to transform the module specific data types to the checkpointing data types. If the original and target representations are known, these transform functions can be developed during the initial stages and made available as libraries.

In addition to this transformation, there could be other instances where data or type transformation is needed. There might be multiple representations being used in the checkpointing server itself: the representation exposed to outside modules might differ from the internal representation being used. As a performance optimization tactic, the checkpointing server might have several different internal representations of the same abstract types. In such a scenario, the internal representation being actively used could be dynamically selected at runtime based on exogenous and internal variables and conditions. Similarly, any application might have access to multiple checkpointing servers allowing it to dynamically choose one that best suits its needs. Such a scenario may arise if the checkpointing servers are being run by external organizations and/or there is a need for multiple checkpointing replicas to offer better reliability. With the availability of multiple checkpoints, an application would be able to choose one that best fit its needs- low latency, low cost, specifications etc. However, this also means that the application needs to have the flexibility to dynamically choose any of the servers and transform its internal representation to that required by the server.

Selecting a checkpoint server and generating detailed checkpointing specifications is only the first step. Once this step has been completed, the individual modules and the application server may proceed to deal with the actual checkpointing process. As part of this process, the application designers must agree on a checkpoint protocol and a communication mechanism for the system. At the very least, this requires that the modules and the checkpointing server agree on the format and timing of the messages being used for checkpointing. To kick off the checkpointing process, the checkpointing server or a controller sends a checkpointing signal. Whether this signal is processed directly by the modules or an intermediary handling the communication level details depends on the scope of the application and the design constraints. Similarly, the response to the checkpointing signal depends on the particular checkpoint protocol being used. In case of a centralized checkpointing server, when the modules receive a checkpointing signal, their response should be to send a module-level snapshot to the checkpointing server. If the checkpoint storage is not centralized, the modules need to follow the checkpoint protocol to ensure that the individual snapshots are mapped to the correct storage location.

A key aspect of the checkpointing protocol is the communication mechanism. This communication protocol is responsible for ensuring that inter-process messages are reliably and correctly transmitted. The frequency and complexity of the message traversal varies with the type of application and the checkpointing protocol. In certain cases, the checkpointing process might require significant amounts of data transfer across the network. However, the representation being generated as part of the checkpoint response might not be the most efficient representation for network traversal. In such a scenario, a network layer manager might be needed to transform the data back and forth between the abstract state representation conforming to the checkpointing specifications and the representation best suited for sending data across the network. Such a transformation would be necessary at all points of communication.

Thus far this chapter has focused on the checkpointing process and the communication protocol during checkpointing. However, before this communication can take place, the application modules have to discover the checkpoint server. This issue of initial discovery is a recurring theme that also comes up in the context of the model game when a player tries to find other players. The mechanism recommended in this thesis approaches the problem in two different ways. In the first approach, the server, or some intermediary, exists at a well-known location allowing the client to establish a communication channel directly. If the server's location is not known in advance, a client can send a multicast message with its request. If a server intercepts the message, it can respond directly and start the communication process. Since sending a multicast request involves more complications and longer delays, having a checkpoint server available at a well-known location is the preferred way to perform server discovery.

Subsequent to the initial discovery, a further mechanism is required to keep checking for server availability. Since the checkpoint server can be a central point of failure, the application requires a guarantee that at least one instance of the checkpointing server is available at any time. Such a task could be handled by a specialized layer in the application or by an outside manager that is responsible for naming, identifying and helping to choose one checkpoint server from several possible options. This external manager can be made responsible for ensuring the availability of a reliable checkpointing server.

Though much of the material discussed to this point has provided no boundary between checkpointing modules and the application being checkpointed, this model needlessly reimplements many of the components central to checkpointing for each application. In the future, the checkpointing capability should be extracted into an independent interface as with other programming components. Additionally, the reliability of the checkpointing functionality could be increased more easily if it were isolated and available as a separate service. Using such an external service, however, raises issues related to security and trustworthiness which we have not discussed in our application. These, we believe, offer avenues for future work.

Bibliography

[1] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems (TOCS), 3(1):63–75, February 1985.

[2] Hofmeister, Christine R. Dynamic Reconfiguration of Distributed Applications, CS-TR-3210, PhD. in the Dept. of Computer Science, University of Maryland, 1993

[3]. Chan, Steven .A Semantic Checkpoint Framework for Enabling Runtime-reconfigurable Applications
Master's Thesis, Dept. of EECS, MIT, 2003

[4]. Bloom, Toby. Dynamic Module Replacement in a Distributed Programming System
PhD in Dept. of EECS, MIT, 1983.

[5] Jeff Magee, Jeff Kramer, Morris Sloman: Constructing Distributing Systems in Conic
I EEE TSE 15, 6, June 1989.

[6]. E.N Elnozahy, Lorenzo Alvisi, Yi-Min Wang, David B. Johnson. A survey of Rollback-Rrecovery protocols in Message-Passing Systems
ACM Computing Surveys, Vol. 34, No. 3, September 2002, pp. 375–408.

[7] M. Herlihy, B. Liskov, A Value transmission Method for Abstract Data Types, ACM Transactions on Programming Languages and Systems, vol2 1982

[8] Object Management Group, Corba Primer
http://www.omg.org/news/whitepapers/seguecorba.pdf

[9] Sathish S. Vadhiyar and Jack J. Dongarra, SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems

[10] João Paulo A. Almeida1,2, Maarten Wegdam1,2, Luís Ferreira Pires1, Marten van Sinderen1, An approach to dynamic reconfiguration of distributed systems based on object-middleware, Proceedings of the 19th Brazilian Symposium on Computer Networks (SBRC 2001), Santa Catarina, Brazil, May 2001.

[11] S Kalaiselvi and V Rajaraman. A survey of checkpointing algorithms for parallel and distributed computers. S¯adhan¯a, 25(5):489–510, October 2000.

[12] Sun Microsystems, Inc. Jini Architectural Overview http://wwws.sun.com/software/jini/whitepapers/, 1999.

[13] Bill N. Schilit, Marvin M. Theimer, and Brent B. Welch. Customizing Mobile Application. In USENIX Symposium on Mobile and Location-independent Computing, pages 129–138, Cambridge, MA, US, 1993.

[14] Anand R. Tripathi, Neeran M. Karnik, Manish K. Vora, Tanvir Ahmed, and Ram D. Singh. Mobile Agent Programming in Ajanta. In Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99), pages 190–197, May 1999.

[15] Steve Ward, Chris Terman, and Umar Saif. Goal-Oriented System Semantics. In MIT LCS Research Abstracts March 2003, pages 139–140. MIT Laboratory for Computer Science, Cambridge, MA, March 2003.