In Architecting Dependable Systems III, 2005

Enabling Safe Dynamic Component-Based Software Adaptation *

Ji Zhang, Betty H.C. Cheng, Zhenxiao Yang, and Philip K. McKinley

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
{zhangji9,chengb,yangzhe1,mckinley}@cse.msu.edu

Abstract. Recomposable software enables a system to change its structure and behavior during execution, in response to a dynamic execution environment. This paper proposes an approach to ensure that such adaptations are *safe* with respect to system consistency. The proposed method takes into consideration dependency analysis for target components, specifically determining viable sequences of adaptive actions and those states in which an adaptive action may be applied safely. We demonstrate that the technique ensures safe adaptation (insertion, removal, and replacement of components) in response to changing external conditions in a wireless multicast video application.
Keywords: safe adaptation, dynamic adaptation, reliability, autonomic computing, middleware, component-based software, dependency analysis.

1 Introduction

Increasingly, computer software must adapt to changing conditions in both the supporting computing and communication infrastructure, as well as in the surrounding physical environment [1]. The need for adaptability is perhaps most acute at the "wireless edge" of the Internet, where mobile devices balance several conflicting and possibly crosscutting concerns, including quality of service on wireless connections, changing security policies, and energy consumption. To meet the needs of emerging and future adaptive systems, numerous research efforts in the past several years have addressed ways to construct adaptable software. Examples include support for adaptability in programming languages [2–4], frameworks to design context-aware applications [5, 6], adaptive middleware platforms that shield applications from external dynamics [7, 8], and adaptable and extensible operating systems [9–11]. In many cases, adaptations involve not only changes to parameters, but reconfiguration of the software structure itself.

Despite these advances in *mechanisms* used to build recomposable software, the full potential of dynamically recomposable software systems can be realized only if the adaptation is performed in a disciplined manner. We use the term "safe adaptation"

* This work has been supported in part by NSF grants EIA-0000433, EIA-0130724, ITR-0313142, and CCR-9901017, and the Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744.

to mean the program maintains its integrity during adaptation. An adaptation is *safe* if (1) it does not violate dependency relationships and (2) it does not interrupt communication either within a component or between components that would potentially yield erroneous or unexpected results. For discussion purposes, we use the term *critical communication segment* to refer to the communication scenarios mentioned in the second part of the definition. Unless adaptive software mechanisms are grounded in formalisms that codify invariants and other properties that must hold during recomposition, the resulting systems will be prone to errant behavior.

This paper describes an approach to ensure safe adaptation in dynamically recomposable systems. This work is part of the *RAPIDware* project, sponsored by the U.S. Office of Naval Research. RAPIDware addresses the design of adaptive middleware for dynamic, heterogeneous environments. Such systems require run-time adaptation, including the ability to modify and replace components, in order to survive hardware component failures, network outages, and security attacks.

Dynamically adaptive software development comprises four major tasks: *Enabling adaptation* makes a program adapt-ready, that is, capable of run-time reconfiguration. *Program monitoring* instruments the program and monitors condition changes in the execution environment. *Decision-making* determines when and how the program should be adapted. *Process management* ensures safe adaptation. Our previous work [3, 12–15] has focused primarily on developing techniques for the first three tasks.

This paper focuses on the fourth task, specifically, ensuring that dynamically adaptive actions are performed safely. Adaptive actions can involve the insertion of a new component, removal of a component, or the replacement of an existing component. Our approach to ensuring safeness during adaptation offers three major features. First, we use invariants to specify dependency relationships among multiple components executing across a single or distributed processes. These dependency relationships enable analysis techniques to determine which components are affected during a given adaptation, and consequently the set of safe states in which dynamic adaptations can take place. Second, our approach provides centralized management of adaptations, thus enabling optimizations to be made when more than one set of adaptive actions can be used to satisfy a given adaptation need. Third, our approach provides a rollback mechanism in case an error or failure is encountered during the adaptation process.

We have applied our safeness techniques to adaptive applications primarily in the mobile computing domain. The remainder of this paper is organized as follows. Background is overviewed in Section 2. In Section 3, we describe the theoretical foundations of our approach. Section 4 describes our proposed approach to safe adaptation in detail, and Section 5 describes its application to a video multicasting system. Section 6 discusses related work, and Section 7 concludes the paper and discusses future directions.

2 Background

Many approaches to compositional adaptation are based on computational reflection [16], which refers to the ability of a program to reason about and alter its own behavior. Typically, reflection is defined to include two parts: introspection (observing internal behavior and state) and intercession (modifying internal behavior and state). Whereas

programming languages such as Java provide support only for introspection, Adaptive Java [3] also supports intercession, thereby enabling the dynamic reconfiguration of software components. The key programming concept in Adaptive Java is that each component offers three interfaces: *invocations* for performing normal imperative operations on the object, *refractions* for observing internal behavior, and *transmutations* for changing internal behavior. An existing Java class is converted into an adaptable component in two steps. The first step, *absorption*, takes place at compile time and produces an *adapt-ready* component. The second step, *metafication*, occurs at run time and equips the adapt-ready component with a set refractions and transmutations. The refractive and transmutative interfaces, respectively, can be used to sense internal state and effect changes to internal structure [17, 18].

We have used Adaptive Java to develop several adaptable components, including *MetaSockets* [12], which are used in the illustrative example described in Section 5. MetaSockets are constructed from the regular Java Socket and MulticastSocket classes, however, their internal structure and behavior can be modified at run time in response to external conditions. MetaSocket behavior can be adapted through the insertion and removal of *filters* that manipulate the passing data stream. For example, filters can perform encryption, decryption, forward error correction, compression, and so forth. In order to maintain a separation of concerns between the original program and the code responsible for adaptation, we applied an aspect-oriented approach to dynamic adaptation [13], where we wove in code containing MetaSockets to make programs adapt-ready for adaptations at run time.

Safe adaptation is important even in situations where the recomposition is relatively constrained. Imagine a distributed application built upon a substrate of MetaSockets, with different ones adapting to different sets of concerns. Guaranteeing that the adaptations are conducted in a consistent manner that will not corrupt the application behavior can be a challenging task. In the remainder of this paper, we describe our approach to addressing this issue.

3 Theoretical Foundations for Safe Adaptation

The adaptations we consider here are component insertion, removal, replacement, and combinations thereof. A component-based software system can be modeled as a set of communicating components running on one or more processes. Components are considered to be communicating as long as there is some type of interaction, such as message exchange, function calls, IPC, RPC, network communication, and so on. A *communication channel* is the facility for communication, such as a TCP connection, an interface, etc. Communication channels are directed. A two-way communication between two components is represented with two channels with traffic traversing in opposite directions. A component can communicate with another as long as there exists a path of one or more channels connecting these two components.

In general, communication among components can be decomposed into multiple non-overlapping communication segments of various granularity. A coarse-grained segment can be divided into multiple finer-grained segments. For example, the communication between a video server and a video client can be divided into multiple trans-

mission/receive sessions; each session can be divided into multiple frames, where each frame can be divided into multiple packets.

Communication can be either local or global. A *local communication* involves components of only one process, such as an ordinary local procedure. *Global communication* involves components from more than one process. A UDP datagram transmission over a network is an example of global communication, involving both sender and receiver processes.

Unsafe adaptation typically involves communication among components. For a given system, if the execution flow can be altered in order to isolate a given component, then valid adaptations of this component, regardless the order in which they are performed, should not affect the correctness of the system. On the other hand, adaptive actions involving communicating components may disrupt normal functional communication between the adapted component and the rest of the system, thus introducing system inconsistencies.

In a given system, multiple components may collaborate by communicating with each other. We use *dependency relationships* to model these communication patterns. The correct functionality of a component, *c*, may require the correct functionality of other component(s). Absence of other components may disrupt normal functionality of *c*.

Dynamic adaptations may interrupt ongoing communication segments. Communication segments whose interruption may cause errors in the system are termed *critical communication segments*. We use a set of finite sequence of indivisible actions (named *atomic actions*) to model the set of critical communication segments *CCS*. The communication among components in a system is modeled as a (finite or infinite) sequence of *critical communication identifier*, atomic action pairs, where a critical communication identifier is a natural number. Given a communication sequence, S, and a critical communication identifier, *CID*, we can extract from S the sequence of atomic actions with the same *CID*, preserving the relative order, denoted S_{CID}. We say an adaptive system does not interrupt critical communication segments if the communication sequence of the adaptive system is S and for all critical communication *CID*, we have $S_{CID} \in$ *CCS*. Based on the discussion above, we define a *safe dynamic adaptation process* as follows:

Definition: A dynamic adaptation process is ***safe*** *iff*

- It does not violate dependency relationships among components.
- It does not interrupt critical communication segments.

In the following subsections, we present our safe dynamic adaptation process.

3.1 Dependency Relationships

Defining Dependency Relationships. In a given system, if the correct functionality of a component *A* depends on a condition $Cond$ to be true, then we say *A depends* on the condition, denoted as $A \rightarrow Cond$, where "$\rightarrow$" denotes a dependency relationship. The condition takes the form of a logic expression across the components. For example, $A \rightarrow (B_1 \oplus B_2) \cdot C$ means the correct functionality of component *A* requires the

correct functionality of either component B_1 or B_2, and C, where the operator "$\oplus$" represents the logical "xor" operation, and "$\cdot$" represents the logical "and" operation. We use a special type of dependency relationship, *structural invariant*, to specify correct conditions of the system structure. For example, the structural invariant $A \cdot B$ indicates that the correctness of the entire system depends on the correct functionality of both component A and component B.

In a safe adaptation process, the dependency condition of a component should always be satisfied when the component is in its fully operational state. Since dependency relationships are based on communication, if we block the communication channels of a component, then we may temporarily relax the dependency relationships and perform necessary adaptive actions. Before the communication in these channels is resumed, the dependency relationships should be reinforced.

Safe Configurations and Safe Adaptation Paths. A *system configuration* comprises a set of components that work together to provide services. If a dependency relationship predicate *dr* is evaluated to be true when we associate *true* to all components in a configuration, and associate *false* to all components not in the configuration, then we say the configuration satisfies the dependency relationship. If a configuration satisfies all the dependency relationships, then this configuration is considered to be a *safe configuration*, otherwise, it is an *unsafe configuration*. A system can only operate correctly when it is in one of its safe configurations. All safe configurations can be deduced from the dependency relationships and available components.

A system moves from one configuration to another by performing *adaptive actions*. An adaptive action is defined as a function from one configuration to another: *adapt(config*$_1$*) = (config*$_2$*)*, where *config*$_2$ is the resulting system configuration when the adaptive action, *adapt*, is applied to *config*$_1$.

A distributed adaptive action comprises multiple local adaptive actions of individual processes. Each local adaptive action is divided into three parts: *pre-action*, *in-action*, and *post-action*. The pre-action is the preparation operation, such as initializing new components, etc. The in-action alters the structure of the program. The post-action specifies tasks to be performed after the in-action, such as the destruction of old components. Pre-actions and post-actions do not interfere with the functional behavior of the adapting process.

We assume an adaptive action is *atomic* and *isolated*. *Atomicity* of an adaptive action implies that the adaptive action should either not start or run to completion. *Isolation* of an adaptive action implies that the adaptive action is performed without interleaving with other operations, i.e., no other operations take place during the adaptive in-action.

An *adaptation step* is an ordered configuration pair: *step =(config*$_1$*, config*$_2$*)*, where $step$ represents a system configuration transition from *config*$_1$ to *config*$_2$. A safe adaptation process comprises a set of safe configurations connected by a set of adaptation steps. These adaptation configurations and steps together form a *safe adaptation path* that starts from the source configuration of the first step and ends at the target configuration of the last step.

We can construct a *safe adaptation graph* (SAG), where vertices are all safe configurations and arcs are all possible adaptation steps connecting safe configurations. A SAG

can be deduced from available adaptive actions. An adaptation step, ($config_1$,$config_2$), is in the SAG *iff*

- Both $config_1$ and $config_2$ are safe configurations.
- There exists an adaptive action $adapt$, $adapt(config_1)=config_2$.

3.2 Critical Communication Segments

Performing adaptive actions may disrupt communication among components. A safe adaptation process should maintain the integrity of critical communication segments, i.e., it cannot interrupt critical communication segments. The system state in which the adaptation does not interrupt any critical communication segments is called a *global safe state*.

If a communication is local, then the integrity of its segments can be maintained by a local process. A local process is said to be in a *local safe state*, if the adaptive action does not interrupt local critical communication segments. The integrity of global critical communication segments is guaranteed by a *global safe condition*, meaning that the adaptive action does not interrupt global critical communication segments. For example, the global safe condition for a UDP-datagram transmission is that the receiver has received all the datagram packets that the sender has sent, where the transmission of each datagram packet is a critical communication segment.

A system is in its global safe state *iff*

- All the processes are in their local safe states.
- The global safe condition is satisfied.

3.3 Enabling Safe Adaptation

Next, we introduce the basis for our safe adaptation process, and prove the process is safe. Consider the following two statements.

(a) An adaptation process is safe.

(b) The adaptation process is a process that executes according to a safe adaptation path, where each adaptive action is performed in its global safe state.

We claim that (a) and (b) are equivalent.

Proof sketch:

(1) $(b) \rightarrow (a)$

If an adaptation process is performed along a safe adaptation path and each adaptive action is performed in a global safe state, then during the adaptation process, the system is either at a safe configuration or in a transition from one safe configuration to another.

When the system is at a safe configuration, it does not violate dependency relationships (definition of safe configuration). Because no adaptive action is performed, critical communication segments will not be interrupted due to adaptations.

Adaptive actions are performed in global safe states, which implies that no critical communication segments will be interrupted. Adaptations start and end in safe configurations, so dependency relationships will not be violated before and after the adaptive action. Adaptive actions are atomic, thus we can assume there is no intermediate states during an adaptive action. Therefore, dependency relationships are not violated during adaptive actions.

(2) Use proof by contradiction to establish $(a) \rightarrow (b)$

If (b) does not hold, then there are two possibilities: (1) the process is not performed along a safe adaptation path or (2) there is an adaptive action taking place in a state that is not globally safe. In the first situation, there must be a configuration on the adaptation path violating dependency relationships, and therefore, the adaptation process is unsafe. In the second situation, the adaptive action might interrupt a critical communication segment, and thus, the adaptation process is unsafe. Therefore, if (b) does not hold, (a) cannot hold. □

4 Safe Adaptation Process

The safe adaptation method is executed by an *adaptation manager*, typically a separate process that is responsible for managing adaptations for the entire system. The adaptation manager communicates with *adaptation agents* attached to processes involved in the adaptation. An agent receives adaptive commands from the adaptation manager, performs adaptive actions, and reports the status of the local process to the adaptation manager. Communication channels can be implemented to best match the communication patterns of the particular system. For example, both Arora [19] and Kulkarni [20] have used a spanning tree, which is well suited to components organized hierarchically. In contrast, in a group communication system, multicast may be a better mechanism for coordination between the adaptation manager and the agent processes.

Our approach comprises three phases: analysis phase, detection and setup phase, and realization phase. The analysis phase occurs during development time. In this phase, the programmers should prepare necessary information such as determining dependency invariants, specifying critical communication segments, etc. The detection and setup phase occurs at runtime. When the system detects a condition warranting adaptation, the adaptation manager should generate a safe adaptation path. In the realization phase, the adaptation manager and the agents coordinate at runtime to achieve the adaptation along the safe adaptation path established during the previous phase.

4.1 Analysis Phase

At development time, the adaptive software developers should prepare a data structure P, where *P = (S, I, T, R, A)*. S is the set of all configurations. I (*I:S* $\rightarrow$ *BOOL*) is the conjunction of the set of dependency relationship predicates. T is a set of adaptive actions. R (*R: T* $\rightarrow$ *PROGRAM*) maps each adaptive action to its corresponding implementation code in the program, where *PROGRAM* represents the implementation. The reconfiguration is achieved by the execution of the implementation code. We associate a fixed cost to each adaptive action. Factors affecting cost values include system blocking

time, adaptation duration, delay of packet delivery, resource usage, etc. The cost of each adaptive action is defined as a function A (A: $T \rightarrow VALUE$), where *VALUE* represents the cost value.

4.2 Detection and Setup Phase

Once the system detects a condition warranting adaptation, the adaptation manager obtains the target configuration and prepares for the adaptation. This phase contains three steps.

1. **Construct Safe Configuration Set.** Based on the source/target configurations of an adaptation request and dependency relationships, this step produces a set of safe configurations.
2. **Construct Safe Adaptation Graph.** Next, we construct a *safe adaptation graph* (SAG) that depicts safe configurations as nodes and adaptation steps as edges.
3. **Find Minimum Safe Adaptation Path (MAP).** Finally, we apply Dijkstra's shortest path algorithm on the SAG to find a feasible solution with minimum weight, where the weight of a path is the sum of the costs of all the edges along the path.

4.3 Realization Phase

This phase requires the coordination of the adaptation manager and the agents at run time to carry out the actual adaptation according to the safe adaptation path. The adaptation manager should ensure that each adaptive action is performed in its global safe state. We use state diagrams to describe the behavior of each agent and the adaptation manager, respectively.

The state diagram of an agent at each local process is shown in Figure 1, where the Courier font denotes message names. Before an adaptive action is performed, each agent is in a running state. In this state, every component in the process is running in its full operation. When the agent receives a `reset` message, then it moves to a resetting state. The agent performs the local pre-action, and initiates a reset of the process. In the resetting state, the process is only partially operating: Some functionalities related to the adapted component are disabled. When the process achieves its local safe state and the condition required by the global safe condition, the agent performs some actions (such as blocking the process) to hold the process in global safe states, so that the local in-action can be performed safely. Then the agent sends the adaptation manager `reset done` message, after which the process is in a safe state. In this state, the agent will perform its local in-action. When the in-action has finished, the agent sends the adaptation manager `adapt done` message and reaches an adapted state. If this agent is not the only one involved in this adaptive action, then the process needs to remain blocked in an adapted state until it receives a `resume` message. When the agent receives a `resume` message, it knows that all processes have completed their adaptive in-actions, so the process proceeds to a resuming state and the agent attempts to resume the process' full operation. If the process is the only one involved in the adaptive action, then it can directly proceed to a resuming state from the adapted state without blocking. Finally, when the full operation of the process is resumed, the

agent sends the manager a `resume done` message and performs the local post-action of the local adaptive action and returns to the original state, running state.

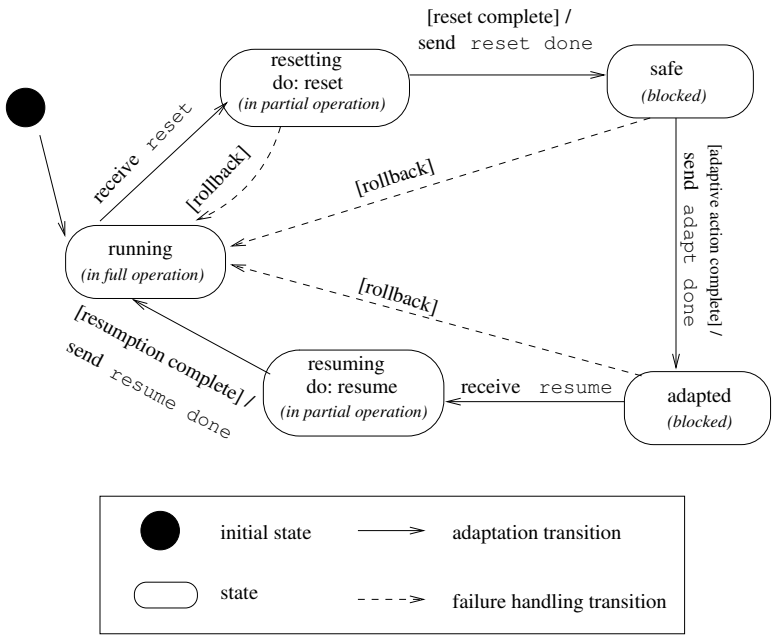

Fig. 1. State Diagram of a Local Process During Adaptation

The state diagram of the adaptation manager of the adaptive system is shown in Figure 2. The adaptation manager starts from a running state where the system is fully operational. When an adaptation request is received by the adaptation manager and a MAP is created after the planning phase, it sends `reset` messages to all the agents. Sending the first `reset` message brings the adaptation manager to an adapting state. In this state, the adaptation manager waits for the `adapt done` message from all agents. When all `adapt done` messages are collected, the adaptation manager proceeds to an adapted state. Then the adaptation manager sends `resume` messages to the agents and the manager proceeds to the resuming state. When the adaptation manager collects `resume done` messages from all agents, it transitions to the resumed state. If there are more adaptation steps remaining in the adaptation path, then the adaptation manager will repeat the traversal of preparing, adapting, adapted, resuming, and resumed states until the system configuration matches the target configuration. When the last adaptation step has finished, the adaptation manager returns to the running state.

4.4 Failure During Adaptation Process

We identify two major types of failures based on our experience. First, if the communication between the manager and the agents is not reliable, then the messages between

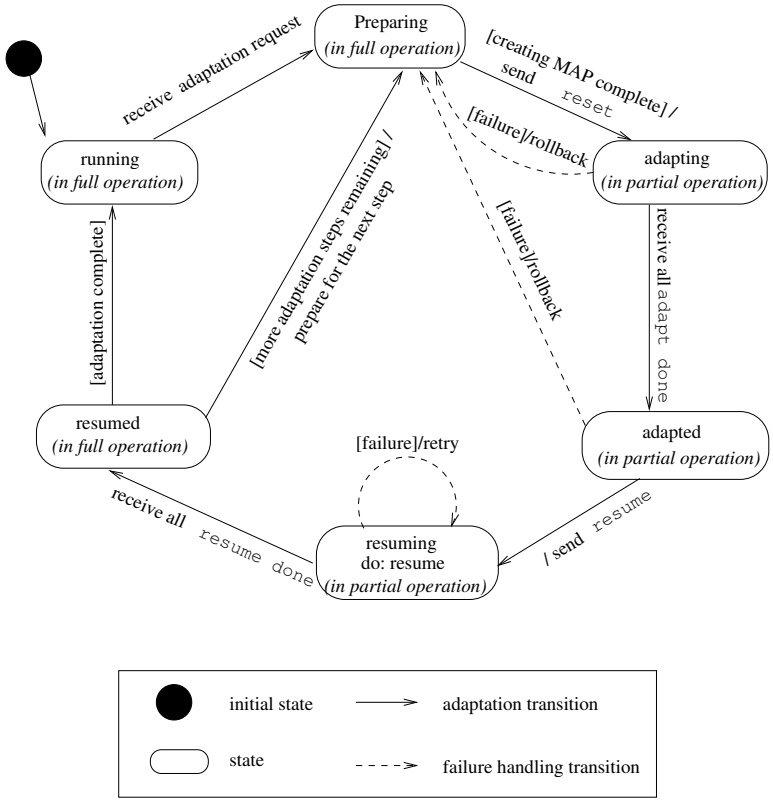

Fig. 2. State diagram of the adaptation manager during adaptation

them may be lost, causing loss-of-message failures. Second, when the agent of a local process receives a `reset` message, the local process may not be able to reach a safe state in a reasonably short period of time, thus causing a fail-to-reset failure. Both types of failures can be detected by a time-out mechanism on the manager.

Loss-of-Message Failure. Loss-of-message failures caused by transient network failures can be handled by several attempts to send the messages. However, loss-of-message failures caused by long-term network failure may cause system inconsistencies if the system does not respond to this type of failures correctly. The general rule for handling loss-of-message failures is that if the failures occur before the manager sends out the first `resume` message, then the adaptation should be aborted. That is, the manager should stop sending any new `reset` and `adapt` messages and all the affected processes should roll back to the state prior to the adaptation. If the failure occurs after the manager has sent out a `resume` message, then the adaptation should run to completion. That is, all the related processes should eventually finish adaptation and resume.

Fail-to-Reset Failure. In some cases, when an agent receives a `reset` message, the local process may be engaged in a long critical communication segment, which may prevent it from reaching a safe state in a reasonably short period of time, thus causing a fail-to-reset failure. If a process cannot reach a safe state after it has received a `reset` message, then the adaptation process should be aborted, and all affected processes should roll back to the state prior to the adaptation.

Failure Handling Strategies. In the event that a failure occurs during an adaptation step, there are two possible outcomes: (1) The adaptation step succeeds and the system reaches the target safe configuration. (2) The adaptation step fails and the system reaches a safe configuration prior to the adaptation. If the adaptation step succeeds, then the manager should continue processing the remaining adaptation steps if there are any. If the adaptation step fails, then the manager has four options: (1) Retry the same step. (2) Try other adaptation paths. (3) Attempt to return to the source configuration. (4) Remain at the current safe configuration and wait for user intervention. We use the combination of all options: The adaptation manager first retries the same step once more. If it still fails, then it tries the second minimum adaptation path from the current configuration to the target configuration. If all possible paths to the target configuration have been tried and have failed, then the adaptation manager tries to return to the source configuration. If this attempt also fails, then the adaptation manager notifies the users and waits for user intervention.

The dashed arrows in Figures 1 and 2 show the failure handling transitions on both the manager and the agents. We claim that the adaptation process is still safe with the presence of failures. During an adaptation step, a rollback is invoked only when no process has been resumed, which ensures that no side effect is produced before the rollback. Otherwise, the adaptation will run to completion, which has the same effect as if the adaptation had had no failures. The interaction between the manager and the agents is similar to the two-phase commit protocol [21] if we combine the safe state with the adapted state in the agents. However, in this work, we consider it clearer to have two separate states. Moreover, our protocol handles multiple adaptation steps where failures may occur at various phases of each step, whereas the two-phase commit protocol only addresses a single adaptation step.

5 Video Streaming Example

We use a video multicasting system to illustrate the safe adaptation process. Figure 3 shows the initial configuration of the application, comprising a video server and one or more video clients. In this example, one client is a hand-held computer (e.g. iPAQ) with a short battery life and limited computing power, and the second client is a laptop (e.g. Toughbook) with reasonable computing power, but limited battery capacity. On the server, a web camera captures video input and a video processor encodes the stream. The encoded video, already packetized, is delivered to the network through a MetaSocket. After traversing a chain of zero or more (encoder) filters, the packets are eventually transmitted on a multicast socket. On each client, the packets are processed by a chain of decoder filters in a receiving MetaSocket. Subsequently, they are passed to

the video processor, where they are decomposed into video frames. Finally the frames are displayed in a video player.

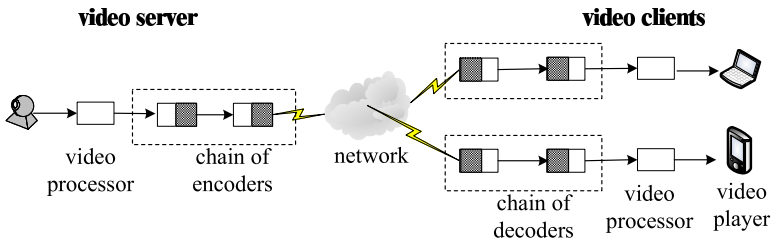

Fig. 3. Configuration of the video streaming application

In this example, two main encryption schemes are available for processing the data: DES 64-bit encoding/decoding, and DES 128-bit encoding/decoding. The sender has two components: E1, a DES 64-bit encoder and E2, a DES 128-bit encoder. The hand-held client has three components: D1, a DES 64-bit decoder, D2, a DES 128/64-bit compatible decoder, and D3, a DES 128-bit decoder. The laptop client has two components: D4, a DES 64-bit decoder and D5, a DES 128-bit decoder. In general, a DES encoder generates DES encrypted packets from plain packets and a DES decoder decrypts the DES encrypted packets. Each decoder implements the "bypass" functionality: when it receives a packet not encoded by the corresponding encoder, it simply forwards the packet to the next filter in the chain. The available adaptive actions are: (1) inserting, removing, and replacing a single encoder or decoder; (2) inserting, removing, and replacing an encoder/decoder pair; (3) inserting, removing, and replacing an encoder/decoder triple. The overall adaptation objective is to reconfigure the system from running the DES 64-bit Encoder/Decoders to running the DES 128-bit Encoder/Decoders to "harden" security at run time. We use a separate process to implement the adaptation manager and attach an agent thread to both the server and the clients, respectively. In this particular application and system architecture, the adaptation manager uses a direct TCP connection to communicate with the agents.

5.1 Safe Adaptation Path and MAP

By analyzing the communication patterns between the encoders and the decoders, we find that the correct functionality of a decoder does not require an encoder, but in order to decode a packet generated by an encoder, there must be a corresponding decoder for each encoder. We have the following invariants, where $\bigotimes$ represents "exclusively select one from a given set of elements".

- System Invariants:
  - **Resource constraint:** $\bigotimes(D1, D2, D3)$
    One of the receivers, the hand-held device, allows only one DES decoder to be in the system at a given time due to computing power constraints.

- **Security constraint:** $\bigotimes(E1, E2)$:
  The sender should have one encoder in the system so that the data is encoded during the adaptation.

– Dependency invariants:
  - $E1 \rightarrow (D1 \vee D2) \wedge D4$
    $E1$ encoder requires the $D1$ or $D2$ decoder to work with the $D4$ decoder.
  - $E2 \rightarrow (D3 \vee D2) \wedge D5$
    $E2$ encoder requires the $D3$ or $D2$ decoder to work with the $D5$ decoder.

We input source and target configurations to the adaptation manager that uses the dependency relationship expressions to generate the safe configuration set. For brevity and automatic processing purposes, we use a 7-bit vector **(D5,D4,D3,D2,D1,E2,E1)** to represent a configuration: If the corresponding bit is "1", then the component is in the configuration, otherwise, it is not. The source configuration is (0100101) and the target configuration is (1010010).

The resulting safe configuration set is shown in Table 1. The adaptive actions shown in Table 2 are input to the adaptation manager. Only related actions are listed. The cost column is packet delay in milliseconds. Note, in order to perform some of the actions (e.g., A6-A9), the server has to be blocked until the last packet processed by the encoder has been decoded by the decoder(s) on the client(s). As a result, these actions are much more costly than other actions.

bit vector	configuration	bit vector	configuration
0100101	D4,D1,E1	1100101	D5,D4,D1,E1
1101001	D5,D4,D2,E1	1101010	D5,D4,D2,E2
1110010	D5,D4,D3,E2	0101001	D4,D2,E1
1001010	D5,D2,E2	1010010	D5,D3,E2

Table 1. Safe configuration set

The adaptation manager creates the SAG shown in Figure 4 and uses Dijkstra's shortest path algorithm to obtain the shortest path, which in this example, has cost 50 ms: A2, A17, A1, A16, A4.

5.2 Performing Adaptive Actions Safely

The adaptation steps for the safe adaptation path are:

1. **Action A2:** Replace D1 with D2.
2. **Action A17:** Insert D5.
3. **Action A1:** Replace E1 with E2.

Action	Operation	Cost (ms)	Description
A1	$E1 \rightarrow E2$	10	replace E1 with E2
A2	$D1 \rightarrow D2$	10	replace D1 with D2
A3	$D1 \rightarrow D3$	10	replace D1 with D3
A4	$D2 \rightarrow D3$	10	replace D2 with D3
A5	$D4 \rightarrow D5$	10	replace D4 with D5
A6	$(D1, E1) \rightarrow (D2, E2)$	100	A1 and A2
A7	$(D1, E1) \rightarrow (D3, E2)$	100	A1 and A3
A8	$(D2, E1) \rightarrow (D3, E2)$	100	A1 and A4
A9	$(D4, E1) \rightarrow (D5, E2)$	100	A1 and A5
A10	$(D1, D4) \rightarrow (D2, D5)$	50	A2 and A5
A11	$(D1, D4) \rightarrow (D3, D5)$	50	A3 and A5
A12	$(D2, D4) \rightarrow (D3, D5)$	50	A4 and A5
A13	$(D1, D4, E1) \rightarrow (D2, D5, E2)$	150	A1 and A10
A14	$(D1, D4, E1) \rightarrow (D3, D5, E2)$	150	A1 and A11
A15	$(D2, D4, E1) \rightarrow (D3, D5, E2)$	150	A1 and A12
A16	$-D4$	10	remove D4
A17	$+D5$	10	insert D5

Table 2. Adaptive actions and corresponding cost.

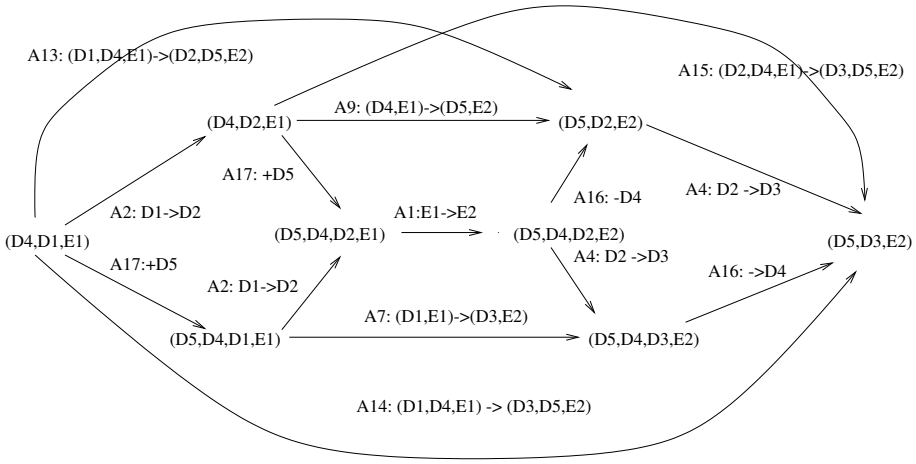

Fig. 4. Safe Adaptation Graph

4. **Action A16:** Remove D4.

5. **Action A4:** Replace D2 with D3.

Step (1), Action $A2$, only involves the process running the MetaSocket on the hand-held. The adaptation manager sends a `reset` message to the agent for the hand-held

device. The global safe state of this action is the same as the local safe state of the device: the DES decoder is not decoding a packet. When the agent receives the `reset` message, it sets a "resetting" flag in the MetaSocket. When the decoder finishes decoding a packet, it checks the "resetting" flag. If it is set, then it notifies the agent and blocks itself. At which point, the agent sends a `reset done` message to the adaptation manager and performs the $(A2 : D1 \rightarrow D2)$ action. Then it sends an `adapt done` message to the adaptation manager. When the adaptive action is done, the agent directly resumes the hand-held's full operation and sends a `resume done` message to the hand-held. Other steps (2-5) can be performed in a manner similar to that used in step (1).

6 Related Work

While there have been numerous techniques developed to support dynamic adaptation, we limit our review to work that explicitly addresses the correctness issue of software adaptation. As part of the RAPIDware project, we introduced an aspect-oriented approach to adding dynamic adaptation infrastructure to legacy programs to enable dynamic adaptation [13]. We separated the adaptation concerns from functional concerns of the program, resulting in a clearer and more maintainable design. Also, as part of the RAPIDware project, Kulkarni *et al.* [20] proposed a distributed approach to safely composing distributed fault-tolerance components at run time. In more recent work [22], they introduced a transitional-invariant lattice technique that uses theorem proving techniques to show that during and after an adaptation, the adaptive system is always in correct states with respect to satisfying the transitional-invariants. Their approach, however, does not guarantee the safeness of adaptation in the presence of failures during adaptation. In contrast, our approach employs a centralized manager, which enables global optimization and ensures safeness.

Other dynamic adaptation techniques have also explicitly addressed correctness issues. Kramer and Magee [23] introduced the notion of a *quiescent state* of a component, in which connections and behaviors of the component can be adapted safely. The concept of quiescent state is close to that of local safe state introduced in this paper. The safe adaptation process in our paper also considers other critical factors such as global conditions and safe configurations. Cactus [24] is a system for constructing highly configurable distributed services and protocols. In Cactus, a host is organized hierarchically into layers, where each layer includes many adaptive components. Chen *et al.* [24] proposed a *graceful adaptation protocol* that allows adaptations to be coordinated across hosts transparently to the application. Appavoo *et al.* [25] proposed a *hot-swapping* technique that supports run-time object replacement. In their approach, a *quiescent state* of an object is the state in which no other process is currently using any function of the object. We argue that this conditions is not sufficient in cases where a critical communication segment between two components includes a series of function invocations. Also, they did not address global conditions for safe dynamic adaptation. Amano *et al.* [26] introduced a model for flexible and safe mobile code adaptation, where adaptations are serialized if there are dependencies among adaptation procedures. Their approach supports the use of assertions for specifying preconditions and postconditions

for adaptation, where violations will cancel the adaptation or roll back the system to the state prior to the adaptation. Their work focuses on the dependency relationships among adaptation procedures, whereas our work focuses on dependency relationships among components.

Researchers have also studied architectural description language (ADL)-based approaches to dynamic architectures. Kramer *et al.* developed Darwin [27], a configuration description language that supports two types of dynamic component instantiations: *lazy instantiation* and *direct dynamic instantiation*. They used π-calculus [28] to define the semantics of the elaboration of Darwin programs. In [29], they used FSP to model adaptive systems, and used a *property automaton* to verify the correctness of adaptations. C2 [30] is an architectural style proposed by Taylor *et al.* ArchStudio [31] is a management tool for dynamic C2 style software evolution.

7 Conclusions and Future Work

This paper presented an approach to safe dynamic adaptation; the approach can be applied in conjunction with existing dynamic adaptation techniques. We use a centralized adaptation manager to schedule the adaptation process, which results in a globally minimum solution. We block the newly added components until the system has reached a new safe state and thus avoid unsafe adaptation. We also use timeout and rollback mechanisms to deal with possible failures during the adaptation process to ensure atomicity of adaptive actions.

Our approach may greatly benefit from using formal requirements specifications of adaptations [32]. In our approach, the detection and setup phase and the realization phase are largely automated. The algorithms are carried out by the adaptation manager and the agent programs. However, as with other approaches that use dependency relationships, the developers must specify the dependency relationships and the adaptive actions in the analysis phase. We are investigating techniques that enable automatic generation of dependency relationships from formal software requirements specifications. Currently, the developers identify the critical communication segments and safe states based on the requirements and the design. If the requirements are formally specified and there is a strict correspondence between the requirements and the design, then it is also possible to automatically derive the critical communication segments and safe states from the formal requirements specifications. One promising approach is to use a temporal logic formula to specify the set of critical communication segments of a component [32]. The run-time component states can be monitored and the formula can then be dynamically evaluated. If all the obligations of the formula are fulfilled in a state, then the state can be automatically identified as a safe state.

Scalability is a concern for our technique. Because our technique searches the optimal path in a SAG, the computational complexity may be high when there are numerous adaptive components in the system (exponential to the number of components involved in an adaptation). To handle the complexity, we can divide the adaptive components of a system into multiple collaborative sets where component collaborations occur only within each set. The component adaptation of each set can be handled independently, thereby reducing the complexity. Also, Dijkstra's shortest path algorithm requires the

entire SAG to be generated. However, in many cases, only a small fraction of the graph is actually related to the given adaptation. We are investigating heuristic-based algorithms that performs partial exploration of the SAG, and would therefore reduce the complexity of this step to $O(n^3)$, where n is the number of components in a collaborative set.

References

1. P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *IEEE Computer*, vol. 37, no. 7, pp. 56–64, 2004.
2. V. Adve, V. V. Lam, and B. Ensink, "Language and compiler support for adaptive distributed applications," in *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001)*, 2001.
3. E. Kasten, P. K. McKinley, S. Sadjadi, and R. Stirewalt, "Separating introspection and intercession in metamorphic distributed systems," in *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02)*, 2002.
4. B. Redmond and V. Cahill, "Supporting unanticipated dynamic adaptation of application behaviour," in *Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002.
5. S. Fickas, G. Kortuem, and Z. Segall, "Software organization for dynamic and adaptable wearable systems," in *Proceedings First International Symposium on Wearable Computers (ISWC'97)*, 1997.
6. J. P. Sousa and D. Garlan, "Aura: An architectural framework for user mobility in ubiquitous computing environments," in *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, 2000.
7. G. S. Blair, G. Coulson, P. Robin, and M. Papathomas, "An architecture for next generation middleware," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, 1998.
8. F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. C. Magalhaes, and R. H. Campbell, "Monitoring, security, and dynamic configuration with the dynamictao reflective orb," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, April 2000.
9. B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility safety and performance in the SPIN operating system," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp. 267–283, ACM Press, 1995.
10. J. Appavoo, R. W. Wisniewski, C. A. N. Soules, *et al.*, "An infrastructure for multiprocessor run-time adaptation," in *Proceedings of the ACM SIGSOFT Workshop on Self-Healing Systems (WOSS02)*, November 2002.
11. D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," in *Symposium on Operating Systems Principles*, pp. 251–266, 1995.
12. S. M. Sadjadi, P. K. McKinley, and E. P. Kasten, "Architecture and operation of an adaptable communication substrate," in *Proceedings of the Ninth International Workshop on Future Trends of Distributed Computing Systems (FTDCS '03)*, May 2003.
13. Z. Yang, B. H. C. Cheng, K. Stirewalt, M. Sadjadis, J. Sowell, and P. McKinley, "An aspect-oriented approach to dynamic adaptation," in *Proceedings of the ACM SIGSOFT Workshop on Self-Healing Systems*, 2002.

14. S. M. Sadjadi and P. K. McKinley, "ACT: An adaptive CORBA template to support unanticipated adaptation," in *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, (Tokyo, Japan), March 2004.
15. S. M. Sadjadi, P. K. McKinley, B. H. Cheng, and R. K. Stirewalt, "TRAP/J: Transparent generation of adaptable java programs," in *Proceedings of the International Symposium on Distributed Objects and Applications*, (Agia Napa, Cyprus), October 2004.
16. P. Maes, "Concepts and experiments in computational reflection," in *Proceedings of the ACM Conerfence on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 147–155, December 1987.
17. P. K. McKinley, E. P. Kasten, S. M. Sadjadi, and Z. Zhou, "Realizing multi-dimensional software adaptation," in *Proceedings of the ACM Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN),* held in conjunction with the *16th Annual ACM International Conference on Supercomputing*, (New York City), June 2002.
18. P. K. McKinley, S. M. Sadjadi, and E. P. Kasten, "An adaptive software approach to intrusion detection and response," in *Proceedings of the 10th International Conference on Telecommunication Systems, Modeling and Analysis*, (Monterey, California), October 2002.
19. A. Arora and M. G. Gouda, "Distributed reset," *IEEE Transactions on Computers*, 1994.
20. S. S. Kulkarni, K. N. Biyani, and U. Arumugam, "Composing distributed fault-tolerance components," in *Proccedings of the International Conference on Dependable Systems and Networks (DSN), Supplemental Volume, Workshop on Principles of Dependable Systems*, pp. W127–W136, June 2003.
21. J. Gray, "Notes on data base operating systems," in *Operating Systems, An Advanced Course*, pp. 393–481, Springer-Verlag, 1978.
22. S. Kulkarni and K. Biyani, "Correctness of component-based adaptation," in *Proceedings of International Symposium on Component-based Software Engineering*, May 2004.
23. J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Trans. Softw. Eng.*, vol. 16, no. 11, pp. 1293–1306, 1990.
24. W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting, "Constructing adaptive software in distributed systems," in *Proc. of the 21st International Conference on Distributed Computing Systems*, (Mesa, AZ), April 16 - 19 2001.
25. J. Appavoo, K. Hui, C. A. N. Soules, *et al.*, "Enabling autonomic behavior in systems software with hot swapping," *IBM System Journal*, vol. 42, no. 1, p. 60, 2003.
26. N. Amano and T. Watanabe, "A software model for flexible and safe adaptation of mobile code programs," in *Proceedings of the international workshop on Principles of software evolution*, pp. 57–61, ACM Press, 2002.
27. J. Kramer, J. Magee, and M. Sloman, "Configuring distributed systems," in *Proceedings of the 5th workshop on ACM SIGOPS European workshop*, pp. 1–5, ACM Press, 1992.
28. R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992.
29. J. Kramer and J. Magee, "Analysing dynamic change in software architectures: a case study," in *Proc. of 4th IEEE international conference on configuratble distributed systems*, (Annapolis), May 1998.
30. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins, "A component- and message-based architectural style for GUI software," in *Proceedings of the 17th international conference on Software engineering*, pp. 295–304, ACM Press, 1995.
31. P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proceedings of the 20th international conference on Software engineering*, pp. 177–186, IEEE Computer Society, 1998.
32. J. Zhang and B. H. Cheng, "Specifying adaptation semantics," in *Proceedings of ICSE 2005 Workshop on Architecting Dependable Systems*, (St. Louis, Missouri), May 2005. accepted for publication.