Composing Distributed Fault-tolerance Components[1]

Sandeep S. Kulkarni Karun N. Biyani Umamaheswaran Arumugam
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824 USA

Abstract— **In this paper, we show how a distributed fault-tolerance component can be replaced dynamically. The need for such replacement arises due to the fact that there are often several fault-tolerance components that are suitable for adding fault-tolerance to a fault, and the choice of the component depends upon the environment. Since a distributed fault-tolerance component has a fraction installed at every process, replacing such a component requires that the replacement be done consistently. More specifically, it requires that the dependency among the fractions of the component be handled so that a component fraction is not removed while other component fractions or the underlying application depends on it. We show how the dependency relation among component fractions is correctly handled while ensuring that the component replacement is transparent to the application. As an illustration of different types of dependency relations, we present a message communication example and show how dynamic composition is performed in it. Finally, in our approach, it is also possible to deal with faults that occur during dynamic composition.**

I. Introduction

Modern distributed systems are often subjected to changing user capabilities, faults, and security threats. Hence, these systems need to be adaptive so that they can change their execution for providing optimum performance based on the changing conditions. For example, if the environment reduces the ability of a process to communicate with other processes in a system, e.g., by causing congestion or network failure, the system may need to decrease the communication-intensive part and increase the computing-intensive part. Or, the system may need to change the way of enforcing security based on the threat level.

In this paper, we focus on fault-tolerant systems that adapt to varying environment conditions and the types of faults encountered. A fault-tolerant system needs to deal with its functionality and fault-tolerance. In [1], Arora and Kulkarni have shown that these two concerns can be separated. More specifically, they have shown that a fault-tolerant application can be decomposed into a fault-intolerant application that deals with functionality (in absence of faults) and a set of *fault-tolerance components* (called *detectors* and *correctors*) that deal with fault-tolerance.

In [1], the authors have also shown that if a fault-tolerant system can be designed by *reusing* the existing fault-intolerant system, then it is possible to design a fault-tolerant system by adding detectors and correctors to the existing fault-intolerant system. However, the choice of the fault-tolerance components (detectors and correctors) used in this composition is not unique. In other words, several fault-tolerance components suffice for providing the required fault-tolerance. Moreover, the performance obtained by using one fault-tolerance component is different from the performance obtained by using another fault-tolerance component. Also, the performance of a fault-tolerance component varies with the environment conditions it is subject to. Therefore, to adapt to the changing environment conditions and application requirements, we need to dynamically replace the fault-tolerance component(s) used for providing fault-tolerance.

Dynamic replacement of a fault-tolerance component in a distributed system is further complicated by the fact that the component itself is distributed. In other words, a distributed component consists of *component fractions* and one component fraction is installed at each process in the system. The application process may depend on the component fraction installed at that process and the component fractions may depend on each other. (See Section IV for a detailed example of such dependency.) Thus, if a component fraction is removed while other component fractions or the application process where it is installed depend on it then the result will be unpredictable. Hence, we need to ensure that the dependency among the component fractions and the dependency between a component fraction and the process where it is installed are correctly handled.

In addition to the dependency among component fractions and the fault-intolerant application, we also need to ensure that the component replacement is transparent to the application. One implication of this transparency is that the addition, removal or replacement of a component should be *atomic*, i.e., as far as the application is concerned, the component replacement should appear indivisible. Secondly, the application should continue to execute during the component replacement or any blocking introduced should be minimal (i.e., only when necessary for correctness). In other words, the component replacement should be *minimally blocking*. Finally, component fractions of two different components should not interact as their interaction could lead to an unpredictable result. In other words, the component replacement should be *synchronized*.

Contributions of the paper. With the above motivation, in this paper, we present an approach for providing the

[1] Email: {sandeep,biyanika,arumugam}@cse.msu.edu
Tel: +1-517-355-2387
This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, and a grant from Michigan State University.

ability to replace distributed fault-tolerance components while ensuring *atomicity*, *minimal blocking* and *synchronization*. Our approach is based on the use of *distributed reset* [2], [3] to ensure that these three properties are satisfied and that the dependency among the component fractions and the intolerant application are correctly handled. Also, our approach has the ability to tolerate faults that occur during the component replacement.

Organization of the paper. The rest of the paper is organized as follows: In Section II, we briefly discuss the architecture of the framework used for dynamic composition. We describe the properties of the reset protocol from [2], [3] in Section III. In Section IV, we identify different types of dependency relations for a distributed component. In Section V, we present how the reset protocol is used in our framework to provide dynamic composition of distributed components. We present an example where our framework is used in Section VI. The Java implementation of the example is discussed in Appendix. We describe some of the issues raised by our framework in Section VII. Finally, we describe the related work in Section VIII and conclude in Section IX.

II. Framework Architecture

To provide the context and broad picture where our work on dynamic composition is used, in this section, we describe the architecture of our framework and briefly explain each of its modules. Our framework provides fault-tolerance to an intolerant application by composing it with fault-tolerance components. The composition of fault-tolerance components with an intolerant application can be asynchronous, synchronous or mixed.

In an asynchronous composition, the execution of the fault-tolerance component is not synchronous with the execution of the intolerant application. Typical examples of fault-tolerance components that require asynchronous composition are components that do the backup of a system, components involved in monitoring environment conditions, etc.

In a synchronous composition, the fault-tolerance component is executed synchronously with the intolerant application, i.e., when the intolerant application executes some portion of its code, the fault-tolerance component executes the corresponding portion of its code. We choose method-level synchronization for synchronous composition. In method-level synchronization, the fault-tolerance component executes its method when a certain method of the intolerant application is executed. The *proactive component* (cf. Section VI) is an example of a component that uses synchronous composition. In this example, the decoding and encoding actions of the proactive component executes synchronously with the send and receive actions of the intolerant application.

Finally, in a mixed composition some piece of code is executed synchronously and some piece of code is executed asynchronously. The *reactive component* (cf. Section VI) is an example of a component that uses mixed composition. In this example, the send and receive actions of the reactive component executes synchronously with the send and receive actions of the intolerant application; and the processing of negative acknowledgments is executed asynchronously with the intolerant application.

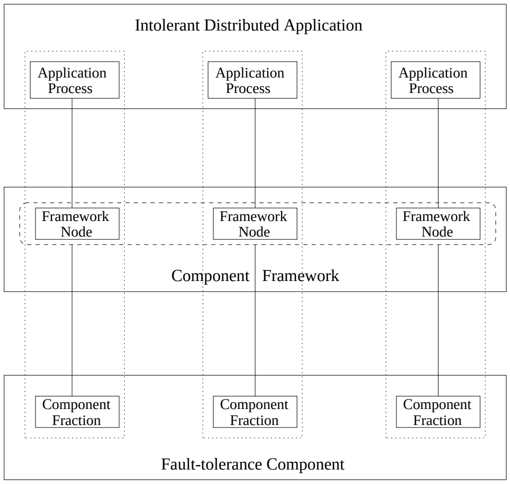

Fig. 1. A Distributed Application Composed With Our Framework

To use our framework, the developer of a fault-tolerance component specifies the methods that are executed synchronously and the methods that are executed asynchronously. The developer of an intolerant application provides guidelines as to what methods of the intolerant application can be executed synchronously with a fault-tolerance component. The developer of the intolerant application may specify individual methods that may be considered for synchronous composition or allow all methods of a certain type (e.g., methods from some class or all public methods) to be synchronized. During composition, the information provided by the developer of the fault-tolerance component and the information provided by the developer of the intolerant application is matched to determine how they should be composed. At runtime, the methods exposed by the intolerant application are trapped to deal with synchronous composition. The part of the fault-tolerance component executed asynchronously is run as a separate thread.

We instantiate a *framework node* at each process in the application (cf. Figure 1). Each framework node consists of a *component manager*, an *adaptation module* and a *reset module* (cf. Figure 2). The component manager performs the addition, removal and replacement of fault-tolerance component fractions. The adaptation module selects the fault-tolerance component based on the environment conditions. In the current implementation, we have the adaptation module at only one node. The reset module ensures that the addition, removal and replacement of fault-tolerance components is atomic, minimally blocking and synchronized. The framework interacts with the intolerant application and the component library. The component library contains detectors and correctors.

Since the main goal of the paper, dynamic addition, removal and replacement of fault-tolerance components, is mostly handled by the reset module, we focus on the reset module in the rest of the paper. We refer the reader to [4] for detailed

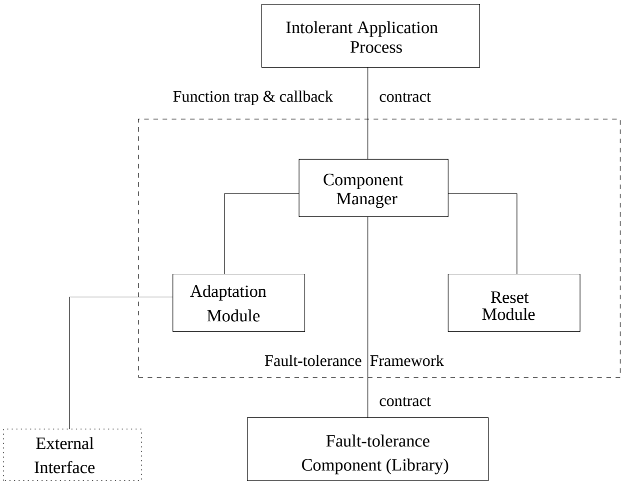

Fig. 2. Architecture of the Framework Node

description of the remaining modules. The format in which the developer of intolerant application specifies exposed methods, the format in which the component developer specifies its requirements, and how they are matched is discussed briefly in Appendix.

Notation. In the rest of the paper, unless specified otherwise, we will use the term component to mean a fault-tolerance component and the term application to mean a fault-intolerant application.

III. Distributed Reset Protocol

In this section, we briefly summarize the reset subsystem of [2], [3] that we use for dynamic addition of distributed components. The reset subsystem can be embedded in an arbitrary distributed system to allow processes to reset the system to a given global state. In the model described in [2], [3], each process consists of an application module and a reset module. The application module at any process may begin the reset operation. The function of a reset module is to (1) reset the state of the application to a state that is reachable from the given global state, and (2) inform the application module when the reset operation is complete.

Each reset operation satisfies the following two properties: (1) Every reset operation is *non-premature*, i.e., if the reset operation completes, then all processes have been reset and the program state is reachable from the given global state, and (2) Every reset operation *eventually completes*, i.e., if an application module at a process initiates a reset operation, eventually the reset module at that process informs the application module that the reset operation is complete. The reset solutions in [2], [3] allow the program computation to proceed concurrently with the reset, to any extent that does not interfere with the correctness of the reset.

To simplify the reset operation, the algorithms in [2], [3] maintain a rooted spanning tree of all non-failed processes. It uses this spanning tree to perform a *diffusing computation* [5] in which each process resets its state. The diffusing computation begins at the root of the spanning tree. The root of the tree resets the state of its local application module and initiates a reset wave that propagates along the tree towards the leaves; whenever the reset wave reaches a process, the process resets the state of its local application module and propagates the reset wave to its children. After the reset wave reaches a leaf, it is reflected as a completion wave towards the root. A process propagates the completion wave to its parent when it receives the completion wave from all its children. The reset is complete when the root receives the completion wave from all its children.

Regarding fault-tolerance, the algorithm in [2] provides stabilizing fault-tolerance to process/channel failures/repairs and transient faults, i.e., starting from an arbitrary state, eventually the program in [2] reaches a state from where future reset operations satisfy non-prematurity and eventual completion. In addition to this stabilizing tolerance, the algorithm in [3] ensures that if only process/channel failures/repairs occur then every reset operation satisfies non-prematurity and eventual completion.

IV. Safe States and Component Dependency

In this section, we describe how the *dependency relation* among component fractions of a distributed component affects their addition and removal.

Safe states of a component. A component fraction of a distributed component cannot be removed arbitrarily as other component fractions or the local application process may depend on it. For example, if component fraction x requires a response from y to continue then removal of y can lead to incorrect functioning, e.g., deadlock, of x. Likewise, the application process where x is installed may be dependent on x. To deal with these problems, we introduce the notion of a safe state. The safe state of a component fraction is further classified as a *global safe state* and a *local safe state*. A component fraction is in a global safe state if (1) no other component fraction depends on it, and (2) the application process where that component fraction is installed does not depend on it. Thus, if a component fraction is in a global safe state, then it is safe to remove it, as its removal will not affect any other component fractions or the application. A component fraction is in a local safe state, if the application can be blocked safely at the process where the component fraction is installed. When a component fraction is in a local safe state, other component fractions may depend on it. However, in a local safe state, the application process at the component fraction does not depend upon the current state of the component fraction. Hence, the application process can be blocked (from communicating with other processes) until the new component fraction is added at that process. We assume that periodically the component fraction at each process will be in a local safe state.

To identify local safe states and global safe states, each component fraction provides a function, *checkState*, whose return value is *safetoremove* (global safe state), *safetoblock* (local safe state) or *unsafetoremove*. The return value of the *checkState* function at the component fraction j is determined based on the current state of j and on the state information of the other component fractions received by j in the past. When a component fraction is in local safe state or global safe state, the information is propagated to other component fractions. This information can, in turn, allow those component fractions to enter local/global safe states. We explain, in Section V-A, how the reset module uses the *checkState* function during reset.

Dependency relation for a component. Now, we discuss different types of dependency relations that exist for a distributed component and how we deal with those dependency relations. For this discussion, we say that x depends on y if there exists a state where removal of y causes incorrect functioning of x.

1) **No dependency.** This is the simplest case. In this case, there exists no dependency among component fractions and they can be removed independently. Hence, all the component fractions will (eventually) return *safetoremove* when *checkState* is invoked.
2) **Acyclic dependency for removal.** As the name suggests, this case deals with the situation where the dependency relation among component fractions is acyclic. It follows that there is at least one component fraction such that no other component fraction depends on it. This fraction can now be removed as *checkState* for this fraction will return *safetoremove*. The removal of this component fraction will, in turn, enable the removal of other component fractions, and so on.
3) **Acyclic dependency with blocking.** Consider a case where we have two component fractions x and y that are mutually dependent. In other words, x (respectively, y) cannot be removed while y (respectively, x) is still running. Further, assume that the application at x can be blocked and the knowledge that the application at x is blocked enables the removal of y. Now, we could remove the fractions x and y as follows: block application at x, remove y and remove x. In this case, initially, when *checkState* is invoked at x (respectively, y), it will return *safetoblock* (respectively, *unsafetoremove*). Later, at some point after x is blocked, *checkState* at y will return *safetoremove* and subsequently, *checkState* will return *safetoremove* at x. More generally, after introducing the notion of blocking, if the dependency relation among component fractions becomes acyclic, then the corresponding component fractions can be removed as in case 2.
4) **Cyclic dependency.** Here the component fractions exhibit mutual dependency even after introducing blocking. Hence, they cannot be removed using any of the three cases discussed above. Possible ways to deal with such dependency are as follows:
    - It is likely that the mutual dependency among component fractions does not exist during all the time while the application is running. There may be instances during run-time, when the component fractions do not depend on each other. Hence, we can add/remove component fractions during those instances.
    - Another approach could be to ignore the dependency relation. Although, this approach may fail in general, if the new component is stabilizing fault-tolerant [6] then it will eventually reach a state from where it will work correctly. This approach is presented in [7].

V. Reset-based Composition of Distributed Components

In this section, we first identify the requirements during dynamic addition of distributed components. These requirements also apply during removal and replacement of distributed components. Then, in Section V-A, we show how the reset module in our framework meets these requirements. Finally, in Section V-B, we show how faults that occur during addition, removal and replacement are handled.

Atomicity. When a distributed component is added to an application, we need to ensure *atomicity* of such integration, i.e., all fractions of the distributed component should be installed across the processes of the application or none should be installed. In other words, if an initiating process adds its component fraction, then all other processes should add their component fraction as well.

Minimal blocking. Another challenge is that during the addition of a distributed component, the application should not be blocked. While it is desirable that the addition of a distributed component be entirely non-blocking, it is not always possible to do so due to dependency among component fractions. We, therefore, require that the blocking introduced during the addition be minimal, i.e., blocking should be introduced only when it is necessary to deal with component dependency.

Synchronization. All processes involved in the addition of components cannot add the component fraction at the same time. Hence, we will have a situation where some processes have added the component fraction, while some have yet to add. If a process that has added a component fraction interacts with another that has not then the results can be unpredictable. Therefore, we must ensure that interactions do not cross a *composition-boundary*, i.e., a process that has added a component fraction does not interact with another process that has not added the corresponding component fraction.

A. Using Distributed Reset for Component Replacement

In this section, we discuss replacement of a distributed component; addition and removal being special cases of replacement. For this discussion, assume that the adaptation module at a process, say X, has decided to replace the distributed component. We call X the *initiator*.

To replace the component, the component manager at X generates a *magic number* for the instance of the new component. The magic number is generated by using the initiator ID, the current time at the initiator, and is used to uniquely identify the instance of the new component. The component manager appends the magic number of the component that the application is using in the message header while communicating with component managers at other processes of the application. The component manager at X uses the reset module for changing the distributed component.

Overview of steps in dynamic component replacement. The dynamic component replacement is achieved by two waves: an *initialization wave* and a *replacement wave*. The replacement wave consists of two sub-waves, namely, a *transition wave* and a *completion wave*. The reset module at X initiates the component replacement by sending the initialization wave. In the initialization wave, all processes change to the *transit state* and initialize the component fraction of the new distributed component. Thus, in the transit state, a process has initialized the new component fraction, although it is still using the old component fraction. Upon successful completion of the initialization wave, the reset module at X starts the replacement wave. The replacement wave begins with transition wave from initiator (root) towards leaves. Each process receiving the transition wave invokes the *checkState* function of the component fraction to determine the state of the component fraction. During the transition wave, the processes remove the old component fraction and add the new component fraction depending on the state information returned by the function. After a leaf process has completed the replacement of its component fraction, the transition wave is reflected as the completion wave to its parent. Further, if a non-leaf process has completed the replacement of its component fraction and it has received the completion wave from all of its children, it propagates the completion wave to its parent. The completion wave eventually reaches the initiator X. We now explain the reset waves in detail.

Initialization wave. The reset module at X initializes the reset by sending an initialization wave to all its neighbors. In this wave, the reset module at X communicates information such as the name of the component, the magic number of the component, and the location of the server where the components are available to the reset modules of its neighbors. Each process that receives the initialization wave performs the following tasks:

1. It sets its parent to the first process from which it received the initialization wave, and propagates the initialization wave to all its neighbors except its parent.
2. If a process receives the initialization wave again it informs the sender the identity of its parent.
3. If the process that receives the initialization wave is a leaf, it initializes the new component and sets itself into the transit state, where it is still using the old component while waiting to use the new component. If the process fails to initialize the new component (e.g., if it lacks the required resources), it sets itself into the *error state*. The process then communicates its state information (transit or error) to its parent.
4. When a process has received the transit state information from all its children, it sets itself into the transit state by initializing the new component. If it receives the error state information from any of its children or if it fails to initialize the new component fraction, it sends the error state information to its parent. Eventually, the root process receives the state information (transit or error) from its children. If it receives the error state information from any of its children, it can restart the initialization wave or abandon the component replacement based on the threshold value set for the number of initialization waves that can be initiated. If the component replacement is abandoned, other processes are informed about this so that they can return to the normal state. If the root process receives the transit state information from all its children, it initializes the new component and sets itself into the transit state.

Transition wave. At the successful completion of the initialization wave, the initiator X sends a transition wave to all its neighbors. When a process receives the transition wave, it performs following tasks:

1) It propagates this wave to all its children.
2) It invokes the *checkState* function, which returns one of the three values: *safetoremove*, *safetoblock* or *unsafetoremove*.
   a) If the function returns *safetoremove*, the process removes the old component, starts using the new component and sets itself into the normal state. Further, the magic number for the new component instance is added in the message header of the subsequent messages. All the other processes that participated in the component replacement are now in the transit state. The component managers at the processes that are in the transit state continuously check the magic number of the messages received. If the process has not started using the new component, all the messages that contain the new magic number are buffered. Note that, once an old component is discarded, the component manager does not need to buffer the messages and it can forward all the messages to the new component.
   b) If the function returns *safetoblock*, the component manager blocks the application process at that component fraction. After the component fraction receives information about other component fractions being blocked/removed, eventually, the function *checkState* at the blocking process will return *safetoremove*. Then, we follow case 2a.
   c) If the function returns *unsafetoremove*, it is periodically invoked till it returns *safetoblock* or *safetoremove*, in which case we follow the case 2b or 2a respectively. For efficiency, we call *checkState* when the application and the component synchronize.

Completion wave. The transition wave is reflected back to the initiator (root) as a completion wave. The leaf process sends the completion wave to its parent after it removes the old component fraction and starts using the new component fraction. Any non-leaf process, which completes the component fraction replacement and receives the completion wave from all its children, sends the completion wave to its parent. When the initiator replaces its component fraction and receives the completion wave from all of its children, the component replacement is complete.

Claim. *The atomicity, minimal blocking and synchronization properties are satisfied during component replacement if the component does not exhibit cyclic dependency.*

Proof sketch. The component replacement starts with an initialization wave. If the initialization wave completes unsuccessfully, then none of the component fractions is replaced. After successful completion of the initialization wave, the reset module starts a transition wave. The component fractions are replaced during the transition wave. A completion wave is propagated towards the initiator when the replacement of the component fraction is complete. In our approach, the completion wave is deferred to handle the dependency relations that exist during the component replacement. As long as the dependency is not cyclic dependency, both the transition wave and the completion wave eventually complete ensuring that all processes replace their component fractions. Thus, component replacement is atomic.

The component replacement blocks the application only when the component fraction is being replaced. Once it is safe to remove a component fraction, it is replaced and the application is unblocked. Thus, the component replacement introduces blocking only if it is required to safely replace other component fractions. It follows that component replacement is minimally blocking.

During component replacement, old component fractions do not communicate with new component fractions; such communication would violate the dependency relation. For example, if the old component fraction at process j communicates with the new component fraction at process k, it would mean that the old component fraction at k was removed while the old component fraction at j was still depending on it. Further, if the new component fraction installed at k communicates with j that is using the old component fraction, the component manager at j buffers these messages. When the new component fraction is added at j, the buffered messages are delivered to the new component fraction. Thus, component fractions of different components do not communicate with each other. Therefore, component replacement is synchronized. □

Addition and removal of distributed component. In the above discussion, we considered the replacement of a distributed component. The addition and removal are special cases of replacement. During instantiation of our framework with the application, our framework traps the functions exposed by the developer of the application and transfers the control to the *default component* that simply calls back the trapped function. Now, for addition of a distributed component, we remove the default component and replace it with the new component. In one conservative approach, the *checkState* function can be implemented as follows: Initially, all component fractions of the default component return *safetoblock*. When all the neighbors are blocked, the *checkState* function returns *safetoremove*. For applications where there is two-way communication between neighbors, this implementation of *checkState* ensures minimal blocking. For applications where there is one-way communication between some neighbors, minimal blocking can be ensured if *checkState* at a process returns *safetoremove* without waiting for the status of processes that do not communicate with it.

B. Dealing with Faults during Component Replacement

In this section, we discuss the extension of the approach in Section V-A to deal with faults that occur during addition, removal or replacement. The algorithm discussed in Section V-A is a variation of the intolerant version of the programs in [2], [3]. By treating the algorithm in Section V-A as an intolerant application and using the fault-tolerance components from [2], [3], we can (statically) add fault-tolerance to that program.

If we were to add the fault-tolerance component from [2], the resulting algorithm will ensure that stabilizing fault-tolerance [6] is provided to faults including process/channel failures/repairs and transients. Thus, even if these faults occur, eventually the application will recover to a state from where subsequent component replacements will be atomic, minimally blocking and synchronized. If we were to add the fault-tolerance component from [3], in addition to the stabilizing fault-tolerance to these faults, the resulting algorithm will provide masking fault-tolerance to process/channel failures/repairs. Thus, if only process/channel failures/repairs occur then the component replacement will be always correct. Moreover, if more general faults such as transients occur then the algorithm will recover to a state from where subsequent component replacements will be correct. Since the algorithm in Section V-A can be used as an input to the framework, it follows that any fault-tolerance property that can be added to the reset program can be added to our framework.

VI. Case Study: Message Communication

In this section, we illustrate how we use dynamic component replacement in the context of a message communication application. We have chosen this simple application because it allows us to demonstrate most of the features including the availability of multiple distributed fault-tolerance components, the need for dynamic composition, and dependency among component fractions. For reasons of space, we only explain the abstract version of the intolerant application and the correctors used to provide fault-tolerance in this application. Then, we explain the dependency relation that exists while adding or removing the fault-tolerance components used for this application. We refer the reader to Appendix for Java implementation of this program and how it is modeled in our framework.

A. Abstract Version of Message Communication

In the abstract version of the program, each action is represented in the following form:

$$\langle\text{name}\rangle :: \langle\text{guard}\rangle \longrightarrow \langle\text{statement}\rangle$$

The guard of an action is a boolean expression over the program variables. The statement of an action updates zero or more program variables. An action can be executed only if its guard evaluates to true. To execute an action, the statement of that action is executed atomically.

Variables. For simplicity, in this discussion, we assume that there is only one sender and one receiver. The sender process generates a message m and sends that message to the receiver process. The application maintains the following variables for each message m:

- $\mathit{send\_in\_future}(m)$: the sender will send message m in future.
- $\mathit{intransit}(m)$: message m is in transit from the sender to the receiver.
- $\mathit{received}(m)$: the receiver has received message m.

Actions. The actions of an intolerant message communication application are as follows:

```
sender::   $\mathit{send\_in\_future}(m) \longrightarrow$
      $\mathit{send\_in\_future}(m)$:=false; $\mathit{intransit}(m)$:=true;

receiver:: $\mathit{intransit}(m) \longrightarrow$
      $\mathit{intransit}(m)$:=false; $\mathit{received}(m)$:=true;
```

Invariant. The invariant characterizes the set of all states from where the intolerant application satisfies its specification. Initially, $\forall m :: \mathit{send\_in\_future}(m)$ is true. Subsequently, if $\mathit{send\_in\_future}(m)$ becomes false, then $\mathit{intransit}(m)$ becomes true. If $\mathit{intransit}(m)$ becomes false, then $\mathit{received}(m)$ becomes true. Thus, an invariant of the intolerant application is as follows (Note that this invariant is not unique; a stronger invariant that requires exactly one of these predicates to be true is also acceptable):

```
invariant:: $\forall m :: \mathit{send\_in\_future}(m) \vee$
      $\mathit{intransit}(m) \vee \mathit{received}(m)$;
```

Fault. The fault action, message loss, is represented as:

```
fault:: $\mathit{intransit}(m) \longrightarrow \mathit{intransit}(m)$:=false;
```

Correctors. Several correctors are available to provide fault-tolerance to message loss. The correction predicate, the predicate to which the intolerant application should be restored after the occurrence of faults, of a corrector that provides required fault-tolerance is: $\forall m :: \mathit{invariant}(m)$.

We consider two fault-tolerance components, a *proactive* component and a *reactive* component. The former is based on the idea of *forward error correction* (FEC) whereas the latter is based on the idea of retransmission. Each of these fault-tolerance components consists of component fractions that are installed at the sender and the receiver processes of the intolerant application.

Proactive component. The proactive component sends extra $(n-k)$ parity packets for each group of k data packets. If any data packet gets lost during transmission, the receiver can generate the lost data packet if it receives at least k packets from a group that contained the lost data packet. Thus, the actions of the proactive component are as follows (Note that this corrector is designed for the fault where no more than $(n-k)$ packets are lost from each group.):

```
sender::   $\forall n, k : n, k > 0, n > k$ :
      for each group of $k$ data packets
      send $k$ data packets and $(n-k)$ parity packets;

receiver:: $\neg \mathit{invariant}(m) \wedge$ at least packets $k$
      are received from group containing $n$ packets
      $\longrightarrow \mathit{received}(m)$:=true;
```

Reactive component. The component fraction at the receiver sends a negative acknowledgment for a lost packet. The component fraction at the sender retransmits the packet for which it receives a negative acknowledgment. Thus, the reactive component detects if $\mathit{invariant}(m)$ is false. Subsequently, it satisfies $\mathit{invariant}(m)$ by setting $\mathit{send\_in\_future}(m)$ to true. The action of the *reactive* component is as follows:

```
sender:: $\neg \mathit{invariant}(m) \longrightarrow \mathit{send\_in\_future}(m)$:=true;
```

B. Dependency Relations

Dependency relation for the proactive component. The component fraction at the sender process encodes the data packets so that for each k data packets it generates $(n-k)$ parity packets. The component fraction at the receiver generates k data packets by decoding the group of n packets. We notice that there exists *acyclic dependency for removal* among these component fractions. More specifically, the component fraction at the receiver process is dependent on the component fraction at the sender process. In other words, if the component fraction at the receiver process is removed before the component fraction at the sender process, then the receiver will not be able to decode the encoded packets that it might still receive from the sender process. Also, the component fraction at the sender process cannot be removed while it has processed a partial group of packets.

Based on the dependency relation, we remove the component fraction at the sender before removing the component fraction at the receiver. Moreover, at the sender, *checkState* returns *unsafetoremove* when a partial group of packets is sent. When the component fraction at the sender is not in the middle of a group, the *checkState* at the sender returns *safetoremove*. Finally, the knowledge about the removal of the component fraction at the sender enables the removal of the component fraction at the receiver.

Dependency relation for the reactive component. The component fraction at the receiver sends a negative acknowledgment for a lost packet. The component fraction at the sender retransmits the packet for which it receives a negative acknowledgment. Clearly, the component fractions of the reactive component are mutually dependent. The component fraction at the receiver process cannot be removed before the component fraction at the sender process is removed. Also, the component fraction at the sender process cannot be removed before the component fraction at the receiver process, since a negative acknowledgment from the receiver may be in transit. In this case, the *checkState* at the sender returns *safetoblock*, as the sender process can be blocked from sending messages. The knowledge about the blocking of the component fraction at the sender enables the removal of the component fraction at the receiver. Thus, the dependency relation of the reactive component falls in the category *acyclic dependency with blocking*.

Based on the dependency relation, we first block the application process at the sender. While the application process at the sender is blocked, the component fraction at the sender can still handle the negative acknowledgments sent by the receiver and can retransmit any lost packets. Eventually, the receiver will reach a state where it has recovered all the lost packets. At this point, the component fraction at the receiver can be safely removed. Subsequently, the removal of the component fraction at the receiver will allow the safe removal of the component fraction at the sender.

VII. Discussion

The framework proposed in this paper raises several questions about how it can be used and modified to suit different applications. We discuss some of these questions below.

Who initiates the component change? Can any process initiate the component change?
Any process can initiate the component change. Although in Section V-A, we assumed that only one process initiates the component change, it is possible to extend it so that other processes can also initiate a component change. Towards this end, we use the approach in [2] where the processes are arranged in a tree. In this case, any process that wants to change a component sends its request to the root. The root process then initiates the component change as mentioned is V-A. This approach also takes care of network partitioning where each partition has its own root process that can perform the component change for that partition.

Is the reset-initialization wave necessary? What are its advantages and disadvantages?
As discussed in Section V-A, a reset-initialization wave is used to initialize the components and create a spanning tree. We note that, the reset-initialization wave is not a requirement for our framework. If we assume that all processes already have the component fractions initialized, then we do not need the reset-initialization wave. If the components are changed frequently and all processes can always succeed in installing the new component, then it would be more efficient to remove the reset-initialization wave. However, if we remove the reset-initialization wave then the overhead incurred by the component manager will increase as it has to check the incoming messages in all cases to determine if a component change is in progress. Also in this case, the process will not have an option to abort the component change. Thus, the decision of using reset-initialization wave is a tradeoff in performance and flexibility rather than a requirement.

Can dynamic composition be improved if components being added are backward compatible with the current component?
Yes. If the new component is backward compatible, then we can add the new component without dealing with the dependency relation among the component fractions. This is due to the fact that the new component fraction at a process can interact with the current component fractions at other processes. Note however that, in general backward compatibility is not satisfied. Hence, in many situations, the component fractions of the new component cannot interact with the component fractions of the current component. It is possible to enhance the framework presented in the paper to simplify composition in this special case. We have not considered this issue in this paper since we are mainly interested in providing dynamic composition in cases where the new component and the current component are independently developed, and hence, are not related.

How can our framework be applied in building secure systems?
Although, our current discussion and implementation focused on fault-tolerance components, our approach can be used in other areas where it is possible to identify independent components that need to be dynamically added. The framework presented in this paper could be used to dynamically add components that provide authentication and/or privacy. Also, in [8], it is shown that a security failure is often a result of number of faults. In such a situation, the dynamic addition of fault-tolerance components will be useful in satisfying the required security properties. Moreover, in [9], it is shown that the theory of detectors and correctors can be applied for adding fault-tolerance to Byzantine faults and these faults are often important in context of modeling security threats. Thus, the framework can be used for satisfying security properties for applications where malicious users can be modeled as users suffering from Byzantine faults. In general, we expect that our framework can be applied if the given security property can be decomposed into safety and liveness formalism (from [10]). However, the problem of providing dynamic composition for more general security properties [11] is still open.

VIII. Related Work

Related work that deals with adaptive fault-tolerant systems includes [12], [13]. The software-based architecture of [12] is composed of subsystems and libraries of metaobjects. Common services required for implementing the metaobjects are provided by the subsystems. A library of fault-tolerance strategies consisting of metaobject classes is implemented on top of the corresponding subsystem. For this reason, the programmer developing a library needs to be aware of the underlying subsystem implementation. In [13], Agha et al. describe a language framework for dependable systems by focusing on modularity and composition. In [13], the base objects specify application specific functionality whereas the meta-level objects specify the fault-tolerance protocols. Our framework differs from the approaches in [12], [13] in that our framework deals with the addition and removal of distributed components by considering the dependency relation among component fractions in a distributed component.

Gouda and Herman [7] have previously considered the problem of adding/removing distributed stabilizing fault-tolerance components. By definition, starting from an arbitrary state, a stabilizing component recovers to a state from where its subsequent computation satisfies its specification. Thus, even if one ignores the dependency relation among component fractions, after the addition of a new stabilizing fault-tolerance component, it will recover to a legitimate state. Our framework differs from the approach of [7] in that in their approach, it is possible for some incorrect computation to occur during a component change as they ignore the dependency relation among component fractions. Moreover, our framework can also deal with components that are not stabilizing fault-tolerant.

Chen et al. [14] have presented an adaptation process that consists of change detection, agreement, and adaptive action. Our approach of changing components is orthogonal to the approach in [14]. In [14], either the dependency relation is ignored or is handled implicitly. This can lead to excessive blocking or incorrect results during component change. We explicitly account for any dependency during component change while ensuring minimal blocking. Secondly, in their approach, faults that occur during the change are not considered. Our approach deals with faults that can occur during component change (cf. Section V-B). Unlike in [14], the reset module described in the paper deals only with the adaptive action that replaces the fault-tolerance component. However, the approach in [14] for change detection and agreement can be combined with our work to build adaptive component-based distributed systems. Further, our work on component change can be used in [14] to ensure that the dependency relation is correctly handled.

Examples such as ESS (Electronic Switching Systems) [15] support dynamic addition/removal of components. However, in these examples, the system consists of a set of applications. When a new component is added, old applications continue to run using the old component whereas the new applications will use the new component. When all the old applications terminate, the old component can be removed. Thus, we can view this system as a set of two disjoint systems; one using the old component and the other using the new component. By contrast, in our algorithm, there is only one (long-running) application that needs to change the component dynamically. Hence, we cannot use a solution where the old and new components execute concurrently until the applications using the old component terminate.

IX. Conclusion and Future Work

In this paper, we presented an approach for dynamic composition of distributed fault-tolerance components. The approach was based on the distributed reset protocol of [2], [3]. Our approach satisfies the three properties, namely, atomicity, minimal blocking and synchronization during component addition, removal and replacement. Also, it correctly handles the dependency relation of a component during dynamic composition. Moreover, as discussed in Section V-B, our approach deals with faults that occur during a dynamic composition.

We used the message communication example (cf. Section VI) to illustrate the availability of multiple fault-tolerance components, the need for dynamic composition, and different dependency relations among component fractions. We have also illustrated the use of our approach in Siesta, Simple NEST Application Simulator [16], developed at Vanderbilt University. In this application, the system consists of 50 nodes. We developed fault-tolerance routing components for Siesta and using the approach presented in this paper we showed how to dynamically change these components. Although, for reasons of space, we omitted this application in the paper, we refer the reader to [4] for detailed description of this application and the corresponding fault-tolerance components.

The approach for dealing with dependency relation among component fractions of a fault-tolerance component is developed in the context of general-purpose framework (cf. Section II). The framework is written in Java language. It permits the case where multiple fault-tolerance components are used simultaneously. For example, in the context of the message communication, we have shown how the *proactive* component and the *reactive* component can be hierarchically composed. Our framework also enables the reuse of fault-tolerance components. The software for the framework is available at: `http://www.cse.msu.edu/˜sandeep/software`.

In the example discussed in Section VI, we explained the replacement of a *proactive component* (FEC-based component) with *reactive component* (acknowledgment-based component) and vice-versa. This replacement did not require the transfer of state information from one component to another. In [4], we have shown how to dynamically change a tree-correction component while preserving state information so that the change of a tree correction component preserves the information about the existing tree.

There are several possible extensions to this work. Currently, we preprocess the source code so that the functions exposed by the fault-intolerant application are trapped and each process is composed with the fraction of the default component (cf. Section V). To increase the applicability of the framework, we are currently focusing on using the binary version of the fault-intolerant application.

Acknowledgments. We would like to thank the anonymous reviewers for their comments and suggestions.

References

[1] A. Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems*, pages 436–443, 1998.

[2] A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[3] Sandeep S. Kulkarni and Anish Arora. Multitolerance in distributed reset. *Chicago Journal of Theoretical Computer Science*, 1998.

[4] Karun Biyani. Dynamic composition of distributed components. Master's thesis, Michigan State University, 2003.

[5] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computation. *Information Processing Letters*, 11(1):1–4, 1980.

[6] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[7] M. G. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17:911–921, 1991.

[8] Catherine Meadows. Applying the dependability paradigm to computer security. In *Proceedings of the 1995 New Security Paradigms Workshop*. pub-IEEE, 1996.

[9] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[10] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[11] Dennis Volpano. Safety versus secrecy. In *Static Analysis Symposium*, pages 303–311, 1999.

[12] Jean-Charles Fabre and Tanguy Perennou. FRIENDS: A flexible architecture for implementing fault tolerant and secure distributed applications. In *European Dependable Computing Conference*, pages 3–20, 1996.

[13] G. Agha, S. Frolund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Proceedings of DCCA-3*, pages 197–207, 1993.

[14] W. K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *21st International Conference on Distributed Computing Systems*, pages 635–643, April 2001.

[15] J. J. Kulzer. Systems reliability: A case study of number 4 ESS. *System Security and Reliability, Infotech State of the Art Report*, pages 186–188, 1977.

[16] Akos Ledeczi, Miklos Maroti, and Istvan Bartok. *SIESTA - Simple NEST Application Simulator (Siesta v0.1 r10.31.1)*. Institute for Software Integrated Systems, Vanderbilt University, Available At: http://www.isis.vanderbilt.edu/projects/nest/downloads.asp, October 2001.

APPENDIX

In this section, we discuss how the message communication application and the two components are implemented in Java. The sender process has a function called *send* and the receiver process has a function called *receive*. These functions are exposed by the intolerant program, i.e., the fault-tolerance component can trap these functions to provide the required fault-tolerance. The component manager at the sender process, which intercepted the send function of the intolerant application, invokes the appropriate function of the fault-tolerance component that is specified by the adaptation module.

Contracts. Now, we discuss the contracts that we have defined for this application. We classify contracts into two types: one that can be verified formally, and another that cannot be verified formally. The contract that cannot be verified formally is represented as a document. One such contract states that for every call to a send function made by a sender there should be a corresponding receive call made by a receiver. The contracts that can be verified formally are represented in a meta-application-file and a meta-component-file. The *meta-application-file* contains the contract between our framework and the intolerant application whereas the *meta-component-file* contains the contract between our framework and the fault-tolerance component. There is one *meta-application-file* for each application process and one *meta-component-file* for each component fraction.

A *meta-application-file* is either supplied by the developer of the intolerant application or it can be generated automatically from the intolerant application. We are currently exploring efficient ways of generating this file. The entries in the *meta-application-file* for the intolerant application process (sender) are as follows (The entries for the receiver process are similar):

```
sendClass:java.net.DatagramSocket
sendFunction:send
sendNumOfArguments:1
sendArguments:java.net.DatagramPacket
sendPacketArgumentNumber:0
...
```

The *meta-application-file* specifies the name of the methods that are exposed by the sender process and other information related to these methods. In this case, the method exposed by the sender process is `send`. The details about this method such as its class, number of arguments, etc. are stored in the *meta-application-file*.

A *meta-component-file* is associated with each fault-tolerance component fraction. This file contains entries similar to a *meta-application-file*. It has some parameters that are to be instantiated with appropriate parameters from the *meta-application-file* before installing this component and some parameters that are supplied by the component developer. The entries in the *meta-component-file* for the fraction of the proactive component at the sender process are as follows (The entries for the fraction at the receiver process and for the reactive component are similar):

```
functionName:send
functionClass:java.net.DatagramSocket
functionArguments:java.net.DatagramPacket
componentFunction: fec-send
...
```

In this particular example, during composition the `functionName` is instantiated with *send* from the *meta-application-file*. Similar instantiations are also performed for other entries. The component uses this file to obtain information about the functions of the intolerant application. This information is provided through appropriate instantiation of the parameters of the *meta-component-file* from the *meta-application-file*. The *meta-component-file* also has information about the component that is used by the framework.

The component manager at the sender process traps the *send* function of the sender process and transfers the control to the *fec-send* function of the fault-tolerance component. Similarly, at the receiver process, the component manager traps the *receive* function of the receiver process and transfers the control to the *fec-receive* function of the fault-tolerance component. When the control is transferred from the intolerant application to the component, the component performs the tasks required by FEC and, if necessary, calls the trapped function of the intolerant application to perform the actual send/receive.

The instantiation of parameters of the *meta-component-file* is done by manual matching of the *meta-application-file* and the *meta-component-file*. We save the *meta-component-file* after it has been instantiated with the application related parameters from the *meta-application-file*. Hence, the future use of the *meta-component-file* during the new instantiation of the component would not require any human intervention. We are exploring heuristics that will allow us to do this matching automatically with minimal human intervention.