

# Woodfrog: A Persistence Library for Smartfrog Components

Rodrigo Schmidt\*  
rodrigo.schmidt@epfl.ch

Paul Murray  
pmurray@hp.com

Hewlett-Packard Laboratories  
Filton Road,  
Bristol BS34 8QZ

## Abstract

This paper introduces Woodfrog, a library that provides persistence properties to Smartfrog components. As arctic wood frogs do to survive cold winters, our library “freezes” a component when there is a system failure so that its internal state can be recovered and the execution resumed after the fault is overcome. Moreover, as part of the “thawing” process, references to the recovered component in other system objects are transparently restored. The library is implemented in a modular way with three very distinct tasks: stable state maintenance, component recovery and reference update. All these tasks can be overwritten and implemented using different algorithms, which allows the user to choose mechanisms more suitable to the application being deployed.

## 1 Introduction

In a system built out of commodity hardware and software, different failures may happen during the execution. The simplest one (and probably the least frequent as well) is a server crash, when the whole server goes down and processes running on it are killed. Most failures, however, are due to software problems and their results might be interestingly complex. If the problem is deterministic and persistent (caused by program mistakes known as Bohrbugs [3]), there is not much that can be done without correcting or reconfiguring the piece of software which is causing the problem. On the other hand, if the problem is transient or intermittent (caused by Heisenbugs [3]), it can be temporarily solved by restarting the application, restarting the system, or resorting to a different replica that did not fail. Some problems, specially Heisenbugs, do not necessarily crash the system or application. They may only make the system malfunction by stopping minor services, degrading performance or wasting resources, problems that might only be identified by an external agent responsible for system recovery. A simple action this agent can perform is to force a server or group of servers to reboot. If the system tolerates server crashes, when it resumes normal execution, the transient problems will have been overcome.

Smartfrog is an application management framework developed at HP Laboratories and aimed at helping configuration, deployment and maintenance of complex systems and applications [12]. Briefly, the whole system is defined in terms of cooperating components organized in a tree-like structure, with

---

\*Work done during internship at Hewlett-Packard Laboratories, in Bristol. Currently a PhD student and research assistant at the Swiss Federal Institute of Technology in Lausanne and the University of Lugano, in Switzerland.

each component directly connected to its parent and children. After the components have been implemented following Smartfrog's component model, a special description language is used to describe the system architecture and configuration. A runtime environment can interpret such descriptions and automatically deploy and manage the system throughout its execution. Smartfrog is used to deploy and manage systems with commodity components (e.g., virtual machines, operating systems, application servers) as well as user applications. In both cases, transient problems can happen during the execution and Smartfrog could be responsible for automatically restarting or recovering failed components.

Albeit a powerful tool for deployment and monitoring of distributed applications, Smartfrog is mainly concerned with a normal execution lifecycle. Its default behavior in case of failures just terminates the system and passes the whole recovery responsibility on to the system administrator. This makes system recovery a rather manual task. Moreover, if the application must support temporary absence, a lot of implementation work will be necessary in order to keep Smartfrog functionalities like the standard lifecycle and the component hierarchy. Our library, called Woodfrog, defines a framework to build *recoverable* Smartfrog components, which are able to survive periods of absence (e.g., due to failures) and transparently recover from them. It extends the original lifecycle and keeps the hierarchy of components created despite the occurrence of failures as long as the failed components are able to recover.

There are a number of known approaches to build fault-tolerant distributed applications as those targeted by Woodfrog, from the use of specific fault-tolerant hardware [6] to the implementation of replicated services in which the failure of a single replica does not compromise the whole system's availability [10]. We wanted to use some mechanism that neither relied on specific machinery nor forced the application to have more resources available than would be necessary in order to run it without providing fault tolerance. Therefore, our approach is based on building crash-recovery components which are able to keep some state information on a stable storage that survives failures and use this information for recovery once the failure has been overcome, a technique known generally as *rollback-recovery* [2].

Many scenarios, as those we previously described in this section, may benefit from the recoverable components implemented by Woodfrog. First, it eases the implementation of crash-recovery distributed algorithms under Smartfrog. Second, it allows applications to use system restart/reboot as a form of correcting transient failures without compromising the application structure or lifecycle. Lastly, components or even entire applications may decide to "freeze" themselves during the execution to release resources for certain periods and be sure that the system will be completely recovered and resumed in the future.

## 2 Recoverable component lifecycle

Smartfrog synchronizes the lifecycle of components deployed under the same structure. During a normal execution, all components synchronously undergo the following transitions:

instantiated  $\rightarrow$  initialized  $\rightarrow$  running  $\rightarrow$  terminated

As a result, two components under the same structure can never be on states *instantiated* and *running*, for example. The lifecycle management mechanism orchestrates these synchronized transitions, allowing components to be in consecutive states only during transition periods. Failures or suspicions during the execution make components transition from its current state to the abstract state *failed* and then, if the default termination behavior has not been overwritten, it triggers the termination of the whole structure, forcing components to transition from their current states to *terminated*.

Due to the eventual necessity of building hybrid systems where recoverable and non-recoverable components would take part in the same deployment structure, transparency became an issue in the project. Therefore, we decided to make the lifecycle of recoverable components a superset of the one of normal Smartfrog components. Figure 1 shows in detail how the two lifecycles differ. The left part of the figure depicts the original lifecycle of a single Smartfrog component. As we mentioned before, most transitions are synchronized amongst the components belonging to the same deployment structure. Moreover, we labeled the main transitions with the name of the call made to the component to trigger the state change. For transparency, we decided to make this state transition diagram also valid for recoverable components and simply extended the original state *running* by dividing it into the two internal states depicted on the right.

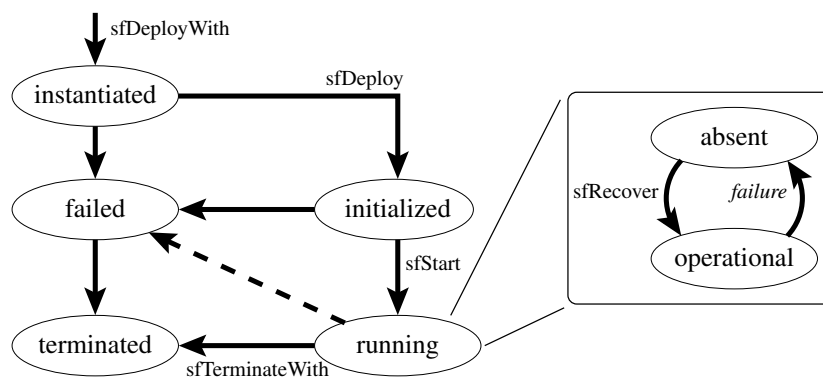


Figure 1: Lifecycle of a recoverable component

In the advent of a failure while a recoverable component is running and operational, instead of transitioning to the *failed* state, the component becomes temporarily absent (in other words, it freezes) until the failure is overcome and the component can be recovered. At this time, it transitions back to the *operational* state. This cycle continues until the component is normally terminated and, therefore, leaves the more general *running* state. One should bear in mind that masking failures in such a way prevents the general transition from *running* to *failed*. For such a reason, Figure 1 shows this transition using a dashed arrow (as this transition exists in a non-recoverable Smartfrog component). However, nothing precludes this behavior from being overwritten by the programmer so that some failures can be considered non-transient and, therefore, trigger application termination.

Many decisions were taken in order to make failure and recovery transparent to the other system components. First, transitions from *operational* to *absent* and back inside a single component are not synchronized with any state transitions in other components. Furthermore, when one component of the system has become absent, the others do not even become aware of it. As we explain later, references

to a recoverable component are transparently wrapped and ordinary calls performed while it is absent are blocked until it recovers. Some specific calls as those concerned with failure suspicion for lifecycle management do not block and return as if the object was still operational to prevent its suspicion.

In the following we explain in more detail the three main modules of the Woodfrog library. The first one is the stable storage, used by a recoverable component to keep its internal state during absence so that it can be recovered in the future. The second module has to do with component recovery and manages the re-deployment of an absent component after the problem that caused its freezing was corrected. Lastly, we show how references to recoverable components kept by other objects in the system are transparently created and updated during the execution.

### 3 Stable storage

It is not possible to build a recoverable Smartfrog component without the notion of a *stable storage*, a data keeper object whose contents survive crashes. In Woodfrog, every recoverable component contains its own stable storage, that is, every component is linked to a stable storage object by composition. The recoverable component uses its stable storage to store all the information necessary for a complete recovery since all its volatile state is lost if a failure occurs. During recovery, a new instance of the component is created containing its previous stable storage. The component then uses the stable storage to recover its volatile state before resuming execution.

#### 3.1 Consistency

Failures are considered to be unpredictable and can happen in the most unexpected moments of the execution, as in the middle of a stable storage update. If this is the case, the failure might render the state kept on stable storage inconsistent and, therefore, prevent a correct recovery. Even if assumptions are made about the occurrence of failures or the atomicity of stable storage operations, it might be the case that the stable state change involves many steps. As a clear example, consider the transition shown in Figure 1 from *initialized* to *running*. At this point of the execution a complete copy of the component state must be written on stable storage to allow the component to recover in case it crashes. This state reflects many operations performed while the component was instantiated and initialized and one cannot assume they can be written to stable storage by a single low-level operation.

Unfortunately, non-atomic modifications are not the only possible cause of inconsistency. Other situations in which it can happen are due to concurrent access internal to the component (e.g., by concurrent threads) or external to it (e.g., by the recovery mechanisms or an incorrectly recovered copy of the component). Work on transactional systems has shown that isolation (concurrency control) and atomicity are the key for consistency [1, 4]. In our specific case, we need a stable storage with three basic features:

- Atomic sequence of actions with all-or-nothing behavior.
- Concurrency control mechanisms for access from different threads inside the component.
- Exclusive update privileges for a single component: the “owner” of the stable storage.

Database management systems (DBMSs) provide a transactional interface satisfying these requirements. However, most common database systems (e.g., MySQL [8], PostgreSQL [9]) are heavy pieces of software and their architecture and interface are more suited to commercial applications. As a result, using them to implement our stable storage would introduce many unnecessary overheads and limitations to the usability of recoverable components. As an example, if each recoverable component has its own different instance of DBMS, not many recoverable components will be allowed to run concurrently on a single machine because of the excessive resource utilization by the databases (mainly memory). If various recoverable components have their stable storages implemented under the same database system (e.g., through different database schemas), the database itself becomes a bottleneck for both performance and reliability.

In order to obtain the transactional interface of a database management system and the lightness of a file-system access library, the best option seems to be the use of an embedded database system to implement a stable storage. Although such databases provide a transactional interface, they have a simpler access semantics (e.g., non-SQL-compliant), assume less amount of data being stored and execute in the same process of the application (with a small footprint). In our prototype, we implemented the stable storage using a stable open source embedded database, namely Berkeley DB Java Edition [11]. However, using it or any other embedded database is not a requirement. In fact, any other type of stable storage can be used with Woodfrog as long as it satisfies the three conditions we mentioned before, which we encapsulate in a Java interface that stable storage classes must implement.

### 3.2 Stable storage usage and implementation

In Woodfrog, each recoverable component has an internal reference to a physical stable storage object as it is shown in Figure 2. This stable storage reference is unique throughout the component's execution, even after its recovery. Moreover, as we require the component to exclusively access it for writing, no two recoverable components are attached to the same stable storage object. This object is responsible for keeping all the component's vital information, that is, the necessary data for it to recover completely after a failure.

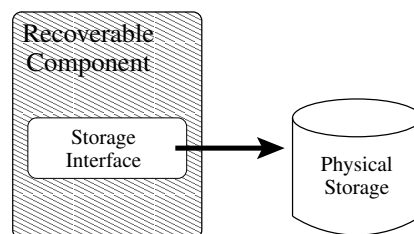


Figure 2: Stable storage access

We call *configuration information* of a component its Smartfrog properties (including both those pre-defined in the configuration language and those created or modified at runtime) together with its local hierarchy information (references to its parent and children). All this data together represents the component's internal Smartfrog state. Indeed, many applications may be able to recover based only on this information and thus, by default, Woodfrog automatically keeps on stable storage all the

configuration information of the respective recoverable component. This is done by writing it on stable storage every time it is changed. Despite the good performance provided by the stable storage object itself, two other things contribute for not having this task as a main source of overhead in the system. First, configuration information is not expected to be modified very often. Second, whenever it happens, only the part of the state actually being changed is updated on stable storage.

The component's application state, that is, the information about the application part of the component, is not automatically written on stable storage. The only way we see this could be transparently done would be by continuously (and at a high frequency) doing Java introspection in the component, which would be prohibitive in terms of execution performance. Moreover, interpreting the internal data structures of a component in order to extract state information is very difficult, if not impossible in some cases (e.g., file access objects). We then, leave this task to the application part of the component, which becomes responsible for saving the necessary state information on stable storage. It does that by directly accessing the component's stable storage object.

Internally to Woodfrog, the stable storage is defined by a Java interface. This allows users to come up with different implementations, according to their needs. A specific Smartfrog property on the recoverable component defines which implementation is used during the execution. The defined interface is quite simple and basically allows writing, modifying and eliminating serializable Java objects on stable storage and grouping sets of operations as transactions. Each stored object is referenced by a name and can be associated to a directory. Directories are just a logical classification for objects. Woodfrog uses it to distinguish hierarchical information from Smartfrog properties. Figure 3 depicts the internal logical structure of a stable storage object.

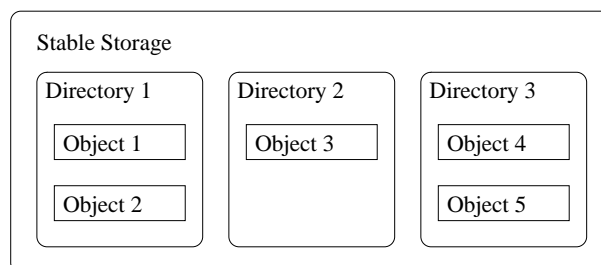


Figure 3: Stable storage structure

### 3.3 Component identification and uniqueness

Since standard (non-recoverable) Smartfrog components are crash-stop by default, identification and uniqueness are not issues that need special consideration during their implementation. A single component instance is created during deployment and, therefore, can be easily identified all over the execution (e.g., by the server it was deployed at and its local address/identification). Recoverable components, however, may have several instances during the execution (one for each incarnation after recovery of a failure) or even migrate from one server to another. Therefore, one cannot use a specific instance to identify a component, and ensuring that there are not two operational instances of the same component running at the same time is not straightforward.

As we decided to make the stable storage unique inside a recoverable component, not shared with other components for updates and the only part of the component that survives crashes, it is an interesting option to consider for component identification. If the stable storage itself is not used to identify a component, we can use some other type of identification (e.g., a randomly generated unique id) and keep it on stable storage. In both cases, the stable storage plays the main role with respect to component identification. Woodfrog, by default, identifies a component based on the reference to its stable storage.

Component uniqueness has to do with not having two independent active instances of a single component working at the same time. If this is not guaranteed, part of the system might end up connected to one instance while the rest of the system is connected to the other component instance. Once more, in standard crash-stop Smartfrog components this does not represent a problem as a single instance of each component is started during deployment. Recoverable components, however, can be wrongly suspected and have another instance started even though the original instance was fully operational but only a little slow. For such cases, there must be some mechanism that either prevents the second instance from starting or makes sure the first instance is killed right after the new one starts. One could put all responsibility onto the recovery subsystem, rendering it more complex and probably making extra assumptions about the system. A simpler approach could provide uniqueness by ensuring exclusive access to some resource that is forcedly used by all instances of the same component, as Woodfrog does. We ensure uniqueness by having the recoverable component to access the stable storage exclusively for writing.

## 4 Component recovery

Component recovery itself is a complex task. Indeed, it comprehends three major problems that must be tackled in order to provide complete fault tolerance based on component restart, to know:

- Identification of a failure
- Creation of a new “surrogate” component
- Installation of previous state in the newly created component

In the following we explain in more detail each one of them and present how we cope with them in our approach.

### 4.1 Identifying a failure

Component recovery is triggered by the identification of a failure. More than that, in Woodfrog it should start only after the failure has been recovered and the stable storage which keeps the component state can be fully accessed. In our first prototype, Woodfrog does that by calling a special component, called “Recovery Agent”, when a computer server containing “frozen” components on its local storage restarts. The recovery agent looks at the locally stored components and tries to open them exclusively. If the local disk is shared and some components have already been restarted in another machine, the recovery agent will not manage to get exclusive access to their stable storage and will assume such a component has already been recovered. All the other frozen components start to be recovered independently.

Different approaches can be used for triggering the recovery procedure, depending on the system architecture. If the stable storage objects are implemented over shared disks, replicated file systems or

a network storage, recovery can be started as soon as the crash of the machines hosting the respective recoverable components is noticed. This can be easily implemented by modifying the dynamic proxy used to do transparent rebinding (later explained on Section 5). A more sophisticated approach could make use of some state monitoring service like Anubis [7] in order to identify failures. After failure identification in such a shared-disk scenario, it suffices to start a recovery agent in a server provided with access to the shared storage system.

## 4.2 Component restart

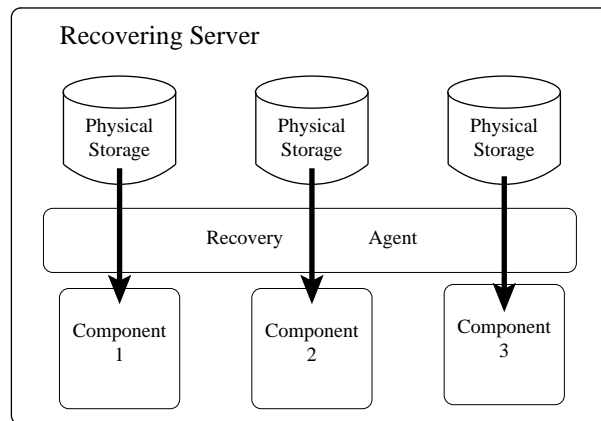


Figure 4: Component recovery after a failed server resumes execution

The recovery agent we mentioned in the previous section is a stateless component whose only responsibility is to restart components previously running on a server that has crashed. The agent is local to the machine in which it is deployed and restarts components based on the stable storage objects kept in that machine, as depicted in Figure 4. The first action a recovery agent takes in order to recover (or “thaw”) a frozen component is to get its basic configuration information (without hierarchy information) from the component’s stable storage. This information is then used to start a new component completely detached from any deployment structure in the system, but containing the Smartfrog properties defined for the recovering component with the latest values they had before it crashed. During initialization of this component, it receives the exclusive control over its stable storage from the recovery agent. After that, the component needs just to restore the part of its state referent to the hierarchy in the original deployment and the application itself. As we use standard Smartfrog methods to create and initialize this new component, changing hierarchy information during the process risks rendering the component inconsistent. Therefore, these steps are performed as the first thing right after initialization.

## 4.3 State installation

After a new “surrogate” component has been created and initialized with the properties’ values stored on stable storage, the recovery agent invokes the method `sfRecover` of the newly created component. The default behavior of this method simply updates the components’ hierarchy state, restoring the latest values prior to the failure. However, this is the method the component should overwrite if some ap-

plication state is also maintained on stable storage and must be recovered as well. After hierarchy and application state have been restored, the component is completely recovered. What is missing now is to let the other objects in the system know that it has recovered and update their references to the new incarnation of the previously crashed component, which we explain in the following section.

## 5 Transparent rebinding

In Smartfrog, components may be referenced by other objects in the system. These references are the way components communicate with each other and are implemented by default through Java RMI references. Conceptually, recovery might not happen on the same machine the failed component was running before. Even if this was assumed to be true, we cannot expect the same reference used to contact a component before it crashed will correctly refer to it after recovery. Moreover, as a project decision, we neither want to change the standard references used by Smartfrog nor want to pay the performance and implementation cost of using a location service every time a recoverable component must be contacted. We want rebinding to be as transparent and efficient as possible, so that the other objects in the system do not have to be aware of whether they are referencing a standard Smartfrog component or a recoverable Woodfrog component.

The way Woodfrog copes with transparent rebinding is by wrapping the RMI references used by Smartfrog with a dynamic proxy. The interface provided by such a proxy is practically the same interface provided by the original RMI reference. If there are no failures, proxy invocations are executed directly over the wrapped RMI reference. If there is a failure, though, its occurrence is internally masked and hidden from the callee. It triggers a rebinding algorithm responsible for updating the internal RMI reference as soon as the called component recovers from the failure.

### 5.1 Dynamic proxy creation

Smartfrog distinguishes between local and remote component references. Component references under the same virtual machine are kept local even when the referenced component has been exported (unless it is obtained through another remote reference<sup>1</sup>). Since we assume unexpected failures that would crash not only a single component but the whole Java virtual machine in which it lies, we have decided not to wrap local references. Woodfrog creates proxies only for remote references and the way it does that is quite simple. Remote references for an object are created when this object is serialized during an RMI call. What Woodfrog does is to overwrite the serialization procedure of recoverable components replacing the original object for a Java Dynamic Proxy [5] wrapping a remote reference to the component (see Figure 5).

Woodfrog also uses Java serialization of properties and internal data structures to write the component's state on stable storage. Therefore, in rewriting the serialization procedure of recoverable components we automatically solve the problem of correctly storing local references to other recoverable components as part of the state. When such references are serialized to stable storage, the serialization

---

<sup>1</sup>This is the case, for example, if the reference is obtained as a result of an RMI call to a remote object. In that case, the returned reference is remote, even though it references a local object.

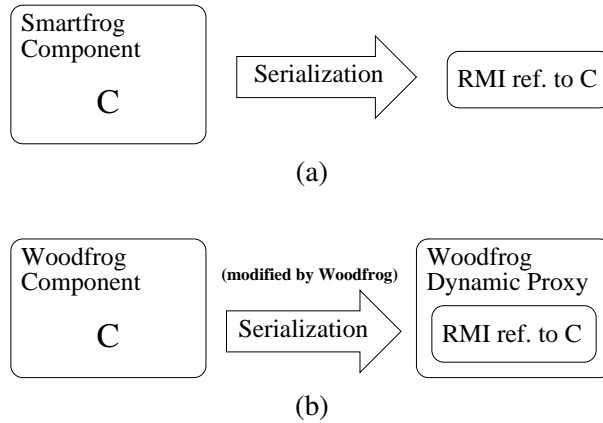


Figure 5: (a) Standard reference creation vs. (b) Woodfrog reference creation

procedure of the recoverable component they reference is invoked and our overwriting of this procedure writes a dynamic proxy referencing the recoverable component. As we explain in detail later, this proxy is able to perform transparent rebinding even after recovery. This is possible because the dynamic proxy we implemented is itself serializable and can be safely stored on stable storage without losing the ability to restore the correct reference to an object after recovery. It might be clear at this point that local references to recoverable components are lost after a crash with subsequent recovery. During recovery, the dynamic proxies stored on stable storage are brought to the memory as part of the recovering component previous state. No local references are recovered. In fact, as we assume components may recover completely independent of each other, they may end up in different virtual machines or even different servers. Trying to recover local references in such a complex scenario is a hard problem which is not tackled in the current prototype.

Internally, a reference proxy must implement a way to find out the updated reference to a component that has failed and recovered. We understand there are many possible implementations, varying on complexity, performance, required assumptions and guarantees. The default discovery algorithm used by Woodfrog is very simple and mainly relies on accessing the stable storage of the referenced component for reading and getting its new reference from there (Section 5.3 explains how this is done in more detail). However, any algorithm can be inserted into the proxy during its creation by the referenced object. During proxy creation, the recoverable component it references is asked by means of an upcall for a class implementing a pre-defined interface with methods to interface the retrieval of a new reference. Should a different algorithm be used, the implementor has to write the code such that a different class is provided during proxy creation.

## 5.2 Remote calls on a recoverable component

If an object makes a call using a proxy reference to a recoverable component and no failures happen, the behavior is almost the same as if a remote reference without proxy was being used. As depicted in Figure 6, the proxy forwards the call to the remote reference it wraps (steps 1-2), the RMI middleware copes with the remote execution of the invoked method (steps 3-4), and the proxy simply relays the result back to the application (steps 5-6). However, if a failure happens to the called component before

it replies the remote call (Figure 7(a)), the proxy must take some action in order to make the failure transparent to the application layer and to automatically restore the reference to the failed component as soon as it recovers. The default implementation of Woodfrog does that by waiting some small amount of time and then accessing the failed component's stable storage for reading to get from there the (possibly new) component's reference, as shown in Figure 7(b). Then the proxy changes its internal reference for the new one and retries the operation that has previously failed. The loop continues until the failed component finally recovers and writes on its stable storage the new reference it has acquired. At this point, since the proxy is able to retrieve the current reference to the previously crashed component, the operation will manage to be executed correctly (as long as the component does not fail again in the middle). We explain in more detail how the proxy is granted access to the component's stable storage in Section 5.3.

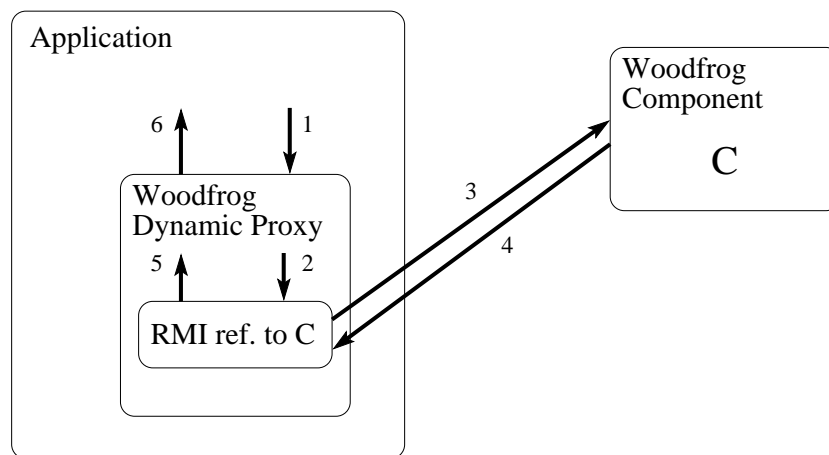


Figure 6: Normal remote calls on a recoverable component

Remote calls on a recoverable component using a proxy are blocked if the called component has failed. They unblock only after the failed component recovers. This project decision was taken to relieve the application from the burden of dealing with exceptions due to component absence and re-executing the failed operation. However, the proxy does not ensure exactly-once semantics for remote calls. It ensures the at-least-once semantics, since failed remote calls are retried until they manage to correctly finish executing. Implementing a general exactly once semantics would need a complex and expensive implementation or strong assumptions about the system. If operations are idempotent, however, the at-least-once semantics ensures a correct execution. If operations are not idempotent, we require that the application do something to ensure a consistent execution in case of failures.

### 5.3 Accessing the referenced component's stable storage

When the proxy has to retrieve a new direct reference to the recoverable component it points to, it resorts to the component's stable storage. This does not mean that stable storages should be highly available, as the absence of connection to the stable storage can be interpreted as if the direct reference internal to the proxy has not changed. However, it is necessary to provide the proxy with a way to access the component's stable storage, which might not be local to the server where the proxy is being

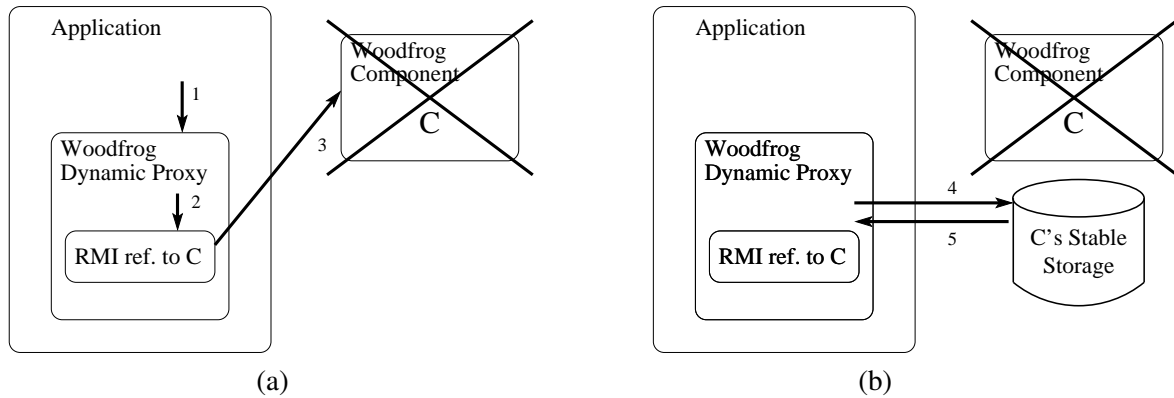


Figure 7: (a) When a call to a recoverable component fails (b) a new reference to the component is obtained from its stable storage

used. As we assume the stable storage is something local to the component using it, which allows good performance during normal execution, the proxy must rely on some remote indirection to access it when the referenced component has failed.

The default discovery algorithm used by the proxy accesses the referenced component’s stable storage by contacting a special component, called storage agent, running at some place with local access to the stable storage. When the proxy is created, the network address of a machine running this service has to be provided (by default, the one running the referenced component). After contacting this machine, the storage agent is located using the naming service embedded in Smartfrog. The storage agent is completely stateless and since it is located through the naming service, it can be crashed and restarted without major problems. Once the proxy manages to contact the storage agent, it asks the service to read the stable storage and return the remote reference stored on it.

## 6 Liveness and blocking

An issue might have been brought out if Smartfrog’s liveness mechanisms were not adapted to work with Woodfrog. Failure detection in Smartfrog is based on `sFPing` calls performed downwards in the deployment tree. Whenever a parent’s ping on a child fails, it suspects the child and, similarly, if a certain amount of time passes without the child being pinged by its parent, the child suspects the parent. We would like to be able to deploy systems where recoverable components coexist with non-recoverable ones and still be able to suspect failed components. This would only be possible if recoverable components reacted to liveness mechanisms differently from non-recoverable ones, masking their transient failures but still propagating liveness calls downwards the deployment tree.

### 6.1 How liveness mechanisms are maintained

If there were no changes on the way recoverable components deal with liveness mechanisms, many problems could happen due to failures. First of all, `sFPing` calls are performed inside a mutual exclusion region (a synchronized part of the code). If the called component is recoverable, this call would be performed through Woodfrog’s reference proxy. As a result, a failure of the called component would

block the callee and preclude it from entering other mutual exclusion regions, including answering the `sfPing` calls it may receive. This domino effect will eventually block all components upwards the deployment tree without necessity. To solve this problem, the proxy takes a different approach when dealing with `sfPing` calls. If there is an error during the call, instead of blocking, the proxy returns as if the recoverable component was still alive. Conceptually, the proxy recognizes that the recoverable component it references is “frozen” but will eventually recover and continue its execution.

Now consider the situation where a recoverable component with some children temporarily fails. If nothing is done, its children will eventually suspect it due to the absence of `sfPing` calls. To be consistent, we also changed the behavior of recoverable components when they are not pinged by their parents. In this case, instead of suspecting them, recoverable components simply assume they are still alive and continue propagating `sfPing` calls downwards the tree. Doing the same thing with non-recoverable components is not possible without changing their original implementation. As we did not want Woodfrog to change the standard behavior of original Smartfrog components, a non-recoverable component do suspect its parent if it is a recoverable component temporarily absent. Moreover, if the default behavior is not changed by the application, such non-recoverable component will eventually terminate because of the “false” suspicion of its parent.

## 6.2 Special care with blocking

In the beginning of the previous section we have shown an example of the dangers of keeping locks when performing remote calls on a recoverable component. The user must bear in mind that such calls may block for an undetermined period of time. If the calling thread holds locks, a big portion of the application might end up blocked. Similarly, assumptions about the execution time of such calls might render the application inconsistent or trigger some unexpected behavior.

## 7 Conclusion and future work

We have presented Woodfrog, a library that provides persistence properties to Smartfrog components. Originally, components were crash-stop, without recovery capabilities in case of failures. Woodfrog makes it possible for components to keep some state on stable storage during failures so that their execution can be resumed after the failure is overcome. Our library performs transparent recovery and automatically reconnects the previously crashed component to the rest of the system. We have explained the principles behind the design of the library and the main algorithms we used to implement it.

We foresee a number of extensions to this first Woodfrog prototype. Different protocols can be used to implement component rebinding during recovery in order to make it more efficient or dynamic. In this case, using Anubis [7] to advertise to the proxies in the system that their component has recovered looks promising. Another possible extension has to do with providing tools to help storing the application state of a component. In this case, if the stored state may not be recent, it might be necessary to coordinate recovery among the system components to put the whole application in a consistent state. Lastly, different implementations of stable storage can be used. It would be interesting to use a highly-available storage that could be used to recover a component immediately after its failure is noticed, without having to wait

for the server in which it was running to restart.

## References

- [1] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [2] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.
- [3] J. Gray. Why do computers stop and what can be done about it? In *Proceedings of the 6th International Conference on Reliability and Distributed Databases*, 1985.
- [4] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [5] Java 1.4.2 Documentation - Dynamic Proxy Classes. <http://java.sun.com/j2se/1.4.2/docs/guide/reflection/proxy.html>.
- [6] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.
- [7] P. Murray. A distributed state monitoring service for adaptive application management. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN-2005)*, pages 200–205, 2005.
- [8] The MySQL Database System. <http://www.mysql.com>, 1995–2005.
- [9] The PostgreSQL Database System. <http://www.postgresql.com>, 1996–2005.
- [10] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
- [11] Sleepycat Software Inc. Berkeley DB Java Edition. <http://www.sleepycat.com>.
- [12] SmartFrog Reference Manual V3.02. <http://www.smartfrog.org>, July 2004.