DR-OSGi: Hardening Distributed Components with Network Volatility Resiliency

Young-Woo Kwon[1], Eli Tilevich[1], and Taweesup Apiwattanapong[2]

[1]Department of Computer Science,Virginia Tech
{ywkwon,tilevich}@cs.vt.edu
[2]National Electronics and Computer Technology Center
taweesup.apiwattanapong@nectec.or.th

Abstract. Because middleware abstractions remove the need for low-level network programming, modern distributed component systems expose *network volatility* (i.e., frequent but intermittent outages) as application-level exceptions, requiring custom manual handling. Unfortunately, handling network volatility effectively is nontrivial—the programmer must consider not only the specifics of the application, but also of its target deployment environment. As a result, to make a distributed component application resilient against network volatility, programmers commonly create custom solutions that are ad-hoc, tedious, and error-prone. In addition, these solutions are difficult to customize for different networks and to reuse across different applications.
To address these challenges, this paper presents a systematic approach to hardening distributed components to become resilient against network volatility. Specifically, we present an extensible framework for enhancing a distributed component application with the ability to continue executing in the presence of network volatility. To accommodate the diverse hardening needs of various combinations of networks and applications, our framework not only provides a collection of hardening strategies, but also simplifies the creation of new strategies. Our reference implementation, built on top of the R-OSGi infrastructure, is called DR-OSGi[1]. DR-OSGi imposes a very low overhead on the hardened applications, requires no changes to their source code, and is plug-in extensible. Applying DR-OSGi to several realistic distributed applications has hardened them with resiliency to effectively withstand network volatility.

Key words: Distributed Component Architectures, Network Volatility, Aspect Oriented Programming, OSGi, R-OSGi

1 Introduction

As the world is becoming more interconnected, our daily existence depends on a variety of network-enabled gadgets. Smart phones, PDAs, GPSs, netbook computers, all run network applications. Many of these gadgets are connected to

[1] Pronounced as "Doctor OSGi" (**D**isconnected **R**emote **O**pen **S**ervice **G**ateway **I**nitiative)

a wireless network such as Wi-Fi. Despite the significant progress made in improving the reliability of wireless networks in recent years, real-world wireless environments are still subject to *network volatility*—a condition arising when a network becomes temporarily unavailable or suffers an outage. Usually the network becomes operational again within minutes of becoming unavailable.

Volatility is a permanent presence of many network environments for several reasons. For one, Wi-Fi networks transmit radio signals, which are volatile, often making it impossible to reach a 100% reliability. Another condition causing network volatility is congestion, which occurs when radio channels interfere with each other or multiple data is transmitted concurrently over the same radio link [8]. Furthermore, wireless networks are rapidly becoming available in emerging markets (e.g., such as in rural or remote areas), which cannot always rely on the existence of an advanced networking infrastructure [29].

Despite its temporary nature, network volatility can prove extremely disruptive for those distributed applications that are built under the assumption that the underlying network is highly-reliable, and network outages are a rare exception rather than a permanent presence. This could happen, for example, when a distributed application, built for a LAN, is later executed in a wireless environment.

Distribution middleware provides a set of abstractions through a standardized API that hide away various complexities of building distributed systems, including the need for low-level network programming. Distributed component systems such as DCOM [14], CORBA CC [18], and R-OSGi [22] expose network volatility as system-level exceptions that are handled by the programmer in an application-specific fashion. Thus, the programmer writes custom exception-handling code that is difficult to keep consistent, maintain, and reuse.

If the underlying network is expected to be volatile during the execution of a distributed system, a consistent strategy can be beneficial for handling the cases of network outages. Manually written outage handling code makes it difficult to ensure that a consistent strategy be applied throughout the application. Since the outage handling code is also scattered throughout the application, it can create a serious maintenance burden. Finally, the expertise developed in handling outages in one distributed application becomes difficult to apply to another application, with a copy-and-paste approach being the only option.

This paper argues that it is both possible and useful to handle network outages systematically, in a consistent and reusable way. Although software architecture researchers have outlined approaches to continue distributed application execution in the presence of network outages, these approaches are difficult to implement, apply, and reuse.

This work builds upon these approaches to define *hardening strategies*, which are exposed as reusable components that can be seamlessly integrated with an extant distributed component infrastructure. These reusable and customizable components can be added to an existing distributed component application, thereby hardening it against network volatility.

As our experimental platform, we use R-OSGi—a state-of-the-art distributed computing infrastructure that enables service-oriented computing in Java. We have created an extensible framework—DR-OSGi—which can harden any R-OSGi application, enabling it to cope with network volatility. DR-OSGi provides programming abstractions for expressing hardening strategies, which can also be reused across applications. The programmer selects a hardening strategy that is most appropriate for a given R-OSGi application and its deployment environment. DR-OSGi then handles all the underlying machinery required to harden the R-OSGi application with the selected strategy.

In our experiments, we have executed several realistic R-OSGi applications in a simulated networking environment to which we injected periodic network outages. By comparing the execution of the original and hardened versions of each application, we have assessed their respective ability to complete the execution, the total time taken to arrive to a result, and the overhead of the hardening functionality. Our results indicate that it is feasible and useful to systematically harden existing distributed component applications with the ability to cope with network volatility. Based on our results, the technical contributions of this paper are as follows:

- A clear exposition of the challenges of treating the ability to cope with network volatility as a separate concern that can be expressed modularly.
- An approach for hardening distributed component applications with resiliency against network volatility.
- A proof of concept infrastructure implementation—DR-OSGi—which demonstrates how existing distributed component applications can be hardened against network volatility.

The rest of this paper is structured as follows. Section 2 introduces the concepts and technologies used in this work. Section 3 describes our approach and reference implementation. Section 4 evaluates the utility and efficiency of DR-OSGi through performance benchmarks and a case study. Section 5 compares our approach to the existing state of the art. Finally, Section 6 presents future research directions and concluding remarks.

2 Background

In the following discussion, we first look at network volatility from the networking perspective. Then we outline the concepts and technologies used in implementing our framework.

2.1 Network Volatility

Modern computing networks are sophisticated multi-component systems whose reliability can be affected by hardware and software failure. These failure conditions include random channel errors, node mobility, and congestion. The reliability of a wireless network can be additionally afflicted by the contention from hidden stations and frequency interference [6, 9].

To improve the performance and reliability of modern networks, researchers have investigated various solutions, including congestion control, error control, and mobile IP. Most of these solutions improve various parts of the actual networking infrastructure. This work, by contrast, is concerned with solutions that treat network volatility as an unavoidable presence to be accommodated in software at the application level.

2.2 Software Components

A software component is an abstraction that improves encapsulation and reusability, thus reducing software construction costs. Typically a component encapsulates some unit of functionality that is accessed by outside clients through the component's interface. Component interfaces tend to remain stable, evolving infrequently and systematically. This reduced coupling between a component and its clients makes it possible to change the component's underlying implementation without having to change its clients. Examples of software component architectures include COM [14], CORBA CC [18], CCA [1], and OSGi [19].

OSGi For our reference implementation, we have chosen a mature software component platform for implementing service oriented applications called OSGi [19]. Among the reasons for choosing OSGi is its wide adoption by multiple industry and research stakeholders, organized into the OSGi Alliance [19]. OSGi is used in large commercial projects such as the Spring framework and Eclipse, which uses this platform to update and manage plug-ins. The OSGi standard is currently implemented by several open-source projects, including Apache Felix, Knopflerfish, Eclipse Equinox, and Concierge[21].

OSGi provides a platform for implementing services. It allows any Java class to be used as a service by publishing it as *a service bundle*. OSGi manages published bundles, allowing them to use each other's services. OSGi manages the lifecycle of a bundle (i.e., moving between install, start, stop, update, and delete stages) and allows it to be added and removed at runtime.

R-OSGi Despite its versatility, OSGi only allows inter-bundle communication within a single host. To support distributed services via OSGi, the R-OSGi distributed component infrastructure was introduced [22]. R-OSGi enables proxy-based distribution for services, providing proxies also as standard OSGi bundles. An R-OSGi distribution proxy redirects method calls to a remote bundle via a TCP channel, supporting both synchronous and asynchronous remote invocations. R-OSGi also provides a distributed service registry, thus enabling the treatment of remote services uniformly with local services.

Thus, R-OSGi introduces distribution transparently, without modifying the core OSGi implementation. It can even enable remote access to an existing regular OSGi bundle, transforming the bundle into a remote service. The transformation employs the concept of *the surrogation bundle*, which registers the service and redirects remote calls to the original bundle.

With respect to network volatility, R-OSGi treats it similarly to other distributed component infrastructures. Specifically, in response to a network disconnection, a client accessing a remote R-OSGi service will receive an exception. The programmer can then write custom code to handle the exception.

2.3 Hardening Strategies to Cope with Network Volatility

When the underlying network fails, a distributed application will typically signal an error to the end user, who can then decide on how to proceed. The user, for example, could choose to check the network connection and restart the application. The purpose of hardening strategies is to enable a distributed application to continue executing when the underlying network becomes unavailable. In a recent publication, Mikic-Rakic and Medvidovic classify disconnected operation techniques as well as how they can be applied to improve the overall system dependability [15]. Next we outline these techniques and discuss how they can be applied to harden a distributed component application to cope with network volatility.

Caching—This strategy employs caching techniques to store a subset of remote data locally, so that it could be retrieved and used by remote service requests when the network becomes unavailable. The effectiveness of this strategy depends strongly on the hit rate of the caching scheme in place. That is, since the size of any cache is always limited, the main challenge becomes to cache the remote data that is most likely to be needed by a service invocation when the network is unavailable. This strategy can in effect fail completely if there is a cache miss.

Hoarding—This strategy prefetches all the remote data needed for successfully completing any remote service invocation. It assumes, however, that data alone is sufficient for invoking a remote service. Unfortunately, this assumption fails for any resource-driven distribution—collocating hardware resources with the code and data they use. For example, a remote sensor has to operate at a remote location from which it is collecting data; hoarding any amount of the sensor's output data will fail to provide up-to-date sensor information upon disconnection. Thus, a hoarding-based strategy can be effective only when computation is distributed for performance reasons, and computation with a given data input yields the same results on any network node. These execution properties are often exhibited by high-performance cluster environments that use distribution to improve performance.

Queuing—This strategy intercepts and records remote requests made to an unreachable remote service. The recorded requests are then replayed when the service becomes available. This technique can only work if the results of a remote call are not immediately needed by the client code (e.g., to be used in an **if** statement). Otherwise, the client code will block, not being able to benefit from this strategy. Queuing is also poorly applicable for realtime applications.

Replication—This strategy maintains a local copy of a remote component. When the remote component becomes unreachable, the local copy is used. If the replicated component is stateful, then the states of the local and remote

copies have to be kept consistent. When the network is available, client requests can be multiplexed to both local and remote copies. Alternatively, a consistency protocol can be used. Upon reconnection, the remote copy has to be synchronized with the local copy. This strategy has the same applicability preconditions as hoarding.

Multi-modal components—This strategy employs several of the strategies above and can apply them either individually, based on some runtime condition, or together, combining some features of individual strategies. For example, both caching and queuing can be used, depending on which remote service method is invoked. Similarly, replication can be applied to remote components while hoarding the data used by the replicated components.

2.4 Aspect-oriented Programming and JBoss AOP

This work aims at treating network volatility resiliency as a distributed cross-cutting concern. A powerful methodology for modularizing cross-cutting concerns is aspect oriented programming (AOP)[13]. We believe that network volatility resiliency is similar to other cross-cutting concerns such as logging, persistence, and authentication—essential functionality, but not directly related to the business logic.

AOP modularizes cross-cutting concerns and weaves them into the application at compile-time, load-time, or runtime. Major AOP infrastructures include AspectJ[12], Spring AOP[25], and JBoss AOP[10]. Some AOP technologies have even been applied to OSGi, including the Eclipse Foundation's AspectJ plug-in and Equinox. For our purposes, we needed to weave in the outage handling functionality at runtime, which typically requires modifying the JVM or rewriting the bytecode. We also needed the ability to modify the parameters of a remote service method. Among the major AOP systems, only JBoss AOP provides all the required capabilities. Another draw of JBoss AOP is that it does not either introduce a new language, thus flattening the learning curve, or changes the JVM, thus ensuring portability.

3 DR-OSGi: Treating Symptoms of Network Volatility

Our reasoning behind the name DR-OSGi—our reference implementation of an infrastructure for systematic handling of network volatility—is our skeptical view of the power of modern medicine. Despite all its impressive accomplishments, modern medicine can only treat some of the symptoms of the majority of known diseases—it cannot eliminate the disease itself. Take common cold as an example. They say that "If you treat a cold, it takes seven days to recover from it, but if you do not, it takes a week." When a cold is concerned, modern medicine can only help eliminate its symptoms, such as fever, sneezing, and coughing, thereby improving the patient's quality of life.

By analogy, we treat network volatility as a disease—an annoying but unavoidable condition that cannot be eliminated. All we want to do is to treat the

symptoms of this disease systematically. By helping the patient (a distributed system) to effectively cope with the symptoms of network volatility (an inability to make remote service calls), we improve the patient's quality of life (QoS).

We next demonstrate our approach by showing how our approach can systematically harden distributed component applications against network volatility. In the following discussion, we first state our design goals, before presenting the architecture of our reference implementation and its individual components.

3.1 Design Objectives

Can any distributed component architecture be effectively hardened against network volatility? In other words, are there any special capabilities a distributed component architecture must provide to make itself amenable to hardening? For our approach to work, we assume that a distributed component architecture can detect and convey to the distributed application the following two scenarios:

1. **A remote service becomes unavailable**—this scenario should be effectively detected by the underlying distributed component architecture, so that an appropriate exception could be raised.
2. **A temporarily unavailable remote service becomes available again**—this scenario assumes that the component architecture does not "give up" trying to reach a remote service, periodically attempting to access it.

To the best of our knowledge, most distributed component architectures can effectively handle the first scenario. However, only advanced distributed component architectures can handle the second one. As a concrete example, R-OSGi employs the Service Discovery Protocol, which periodically attempts to reconnect to a remote service, if the service were to become unavailable. If, for example, a remote service becomes unreachable due to a network outage, the R-OSGi Service Discovery Protocol will keep trying to reach the service until the network connection is restored. It is these advanced capabilities of R-OSGi that convinced us to use this distributed component architecture as our experimentation platform.

Our system, called DR-OSGi, can harden existing R-OSGi applications to become resilient against network volatility. In designing DR-OSGi, we pursued the following goals:

1. **Transparency**—any hardening strategy should not affect the core functionality of the underlying R-OSGi application.
2. **Flexibility**—DR-OSGi should be capable of adding or removing the hardening strategies at any time without having to stop the application.
3. **Extensibility**—DR-OSGi should provide flexible abstractions, enabling expert programmers to easily implement and apply custom hardening strategies.

3.2 Design Overview

The purpose of DR-OSGi is to harden an R-OSGi application with resiliency to cope with network volatility. Thus, to explain the general architecture of DR-OSGi, we start by outlining the fundamental building blocks of R-OSGi. Figure 1 shows that R-OSGi integrates a remoting proxy that redirects service calls to a remote OSGi bundle and also transfers the results of the calls back to the client.

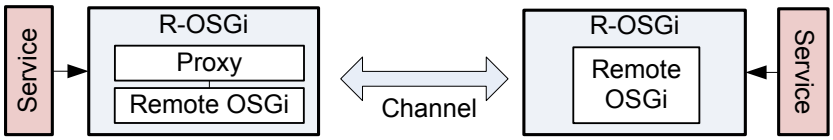

Fig. 1. Initial architecture of R-OSGi.

Since the channels to a remote OSGi bundle use TCP, which provides reliable data transport, packet loss is handled at the transport layer. TCP, however, provides no assistance to deal with network volatility conditions arising as a result of link failure, node mobility, or high congestion. Therefore, to detect network instability or disconnection, an R-OSGi channel uses a timer to block the caller until the service has returned or the timeout has been exceeded. In the case of exceeding the timeout, an exception is thrown. R-OSGi handles such exceptions by having a remote OSGi bundle dispose of the channel and remove all proxies, preventing remote service calls while the network in unavailable. R-OSGi periodically checks whether the network has become available again and, if so, recreates the remoting proxies and channels.

DR-OSGi intercepts the handling of R-OSGi network-related exceptions and the successful completions of its reconnection attempts. Specifically, DR-OSGi handles R-OSGi network-related exceptions by triggering a hardening strategy. The type of the triggered strategy is determined by a programmer-specified configuration. The hardening strategy stops being applied when DR-OSGi intercepts a successful R-OSGi reconnection attempt.

Figure 2 shows how DR-OSGi is integrated into a typical R-OSGi application. DR-OSGi augments an R-OSGi application with a hardening manager and a collection of hardening strategies. The manager and each strategy are encapsulated in separate OSGi bundles.

The hardening manager plugs into an R-OSGi application to intercept the handling of network exceptions and of the successful completions of reconnection attempts. In response to these events, the manager starts and stops the hardening strategies as configured by the programmer.

To integrate the hardening manager with an R-OSGi application without changing the application's source code, we employ Dynamic Aspect Oriented Programming. Because OSGi bundles are deployed at runtime, DR-OSGi has

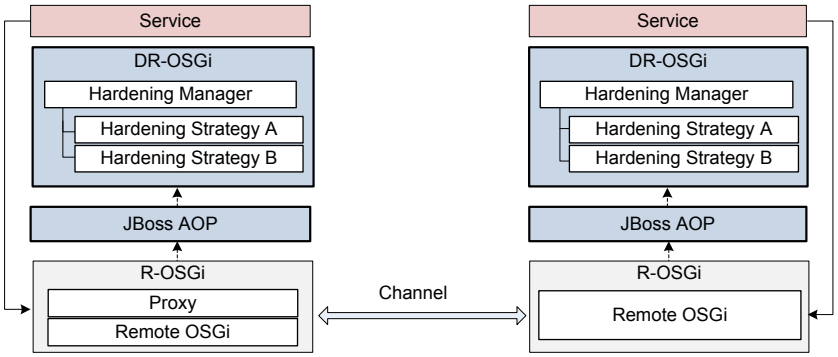

Fig. 2. Hardened architecture.

to be able to interpose the hardening logic dynamically. The dynamic AOP technology that fits our design objectives is JBoss AOP.

3.3 Programming Model

Next we detail the DR-OSGi programming model and demonstrate how it simplifies the creation and deployment of custom hardening strategies. To harden an R-OSGi application, the programmer has to provide a configuration file that specifies which hardening strategy should be applied to which application bundle. The following configuration file specifies that the application bundle "MyBundle" is to be hardened by the strategy implemented in the DR-OSGi-conformant bundle "CachingHardening":

```
RemoteServiceName=org.mypackage.MyBundle
HardeningServiceName=org.otherpackage.CachingHardening
```

The simple syntax of the DR-OSGi configuration files is sufficiently expressive and supports wildcards which can be used to specify that a hardening strategy be applied to multiple bundles. Several hardening strategies can be applied to the same application bundle simultaneously. For example, remote invocations can be both cached and queued when the network is available. The programmer can specify in the configuration file which strategy bundle should be primary (i.e., to be applied first). If, when the network becomes unavailable, the first strategy succeeds, DR-OSGi does not apply the second one.

To implement a hardening strategy, the programmer needs only to implement interface `DisconnectionListener`, which is defined as follows:

```
public interface DisconnectionListener {
  public Object disconnectedInvoke(RemoteCallMessage invokeMessage);
  public Object reconnected(String uri);
  public void remoteInvoke(RemoteCallMessage invokeMessage, Object result);
  public void serviceAdded(String uri);
  public void serviceRemoved(String uri);
}
```

Method disconnectedInvoke is called by DR-OSGi, when R-OSGi detects that the network connection has been lost. Method reconnected is called by DR-OSGi, when R-OSGi manages to successfully reestablish a connection to a remote bundle. Finally, remoteInvoke is called when a remote service method has been successfully invoked.

The implemented class has to be deployed as a regular OSGi bundle, and an entry describing the implementation must be added to the configuration file.

3.4 System Architecture

In the following we discuss the system architecture of DR-OSGi. The key objective of this work is to explore how network volatility hardening strategies can be implemented modularly and applied to an existing distributed component application that may have been written without fault-tolerance capabilities in mind. In other words, we argue that it is possible to treat hardening strategies as reusable software components, which can be developed by third-party programmers and reused across multiple applications.

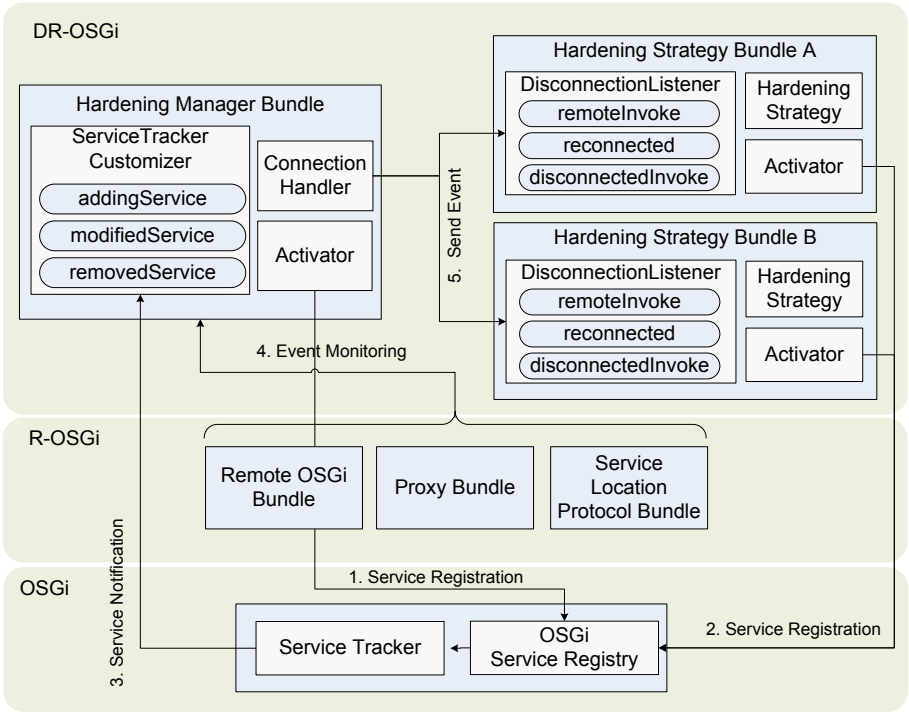

Fig. 3. DR-OSGi Design.

Figure 3 shows how we have designed DR-OSGi, so that it could naturally integrate with the existing OSGi and R-OSGi infrastructures. DR-OSGi makes use of existing OSGi services such as `Service Registration` and `Service Tracker`. Every DR-OSGi component, including the hardening manager and all hardening strategies, register themselves with OSGi, which manages them as standard registered services. This arrangement makes it possible to locate DR-OSGi components using the OSGi `Service Tracker` and load them on demand.

To receive service change events from OSGi, the hardening manager implements the ServiceTrackerCustomizer interface, which is discussed below. In turn, to make it possible for the manager to send the relevant events to hardening strategy bundles, each bundle implements the DisconnectionListener interface. All the lifecycle events in DR-OSGi are triggered by sending and receiving events, with `Service Tracker` and `Service Registration` enabling the hardening manager and hardening bundles to be loosely coupled.

When a new hardening strategy is deployed, OSGi sends an event—addingService—to `Service Tracker`, which then forwards the event to the hardening manager by calling the corresponding ServiceTrackerCustomizer interface method.

```
public interface ServiceTrackerCustomizer {
  public Object addingService( ServiceReference reference );
  public void modifiedService( ServiceReference reference , Object service );
  public void removedService(ServiceReference reference , Object service );
}
```

The hardening manager keeps track of which hardening strategies have been registered and maintains a searchable repository of all the registered strategy bundles.

Weaving in Resiliency Strategies with Aspects To intercept the disconnection/reconnection procedures of R-OSGi, without changing its source code, we use dynamic Aspect Oriented Programming technology, JBoss AOP. The ability to apply aspects dynamically is required due to OSGi loading bundles dynamically at runtime. JBoss AOP makes use of XML configuration files that specify at which points aspects should be weaved. Using AOP enables DR-OSGi to keep its implementation modular and avoid having to modify the source code of R-OSGi.

3.5 Discussion

The hardening approach of DR-OSGi is quite general and can be applied to a variety of distributed components. Although our reference implementation is dependent on R-OSGi and JBoss AOP, DR-OSGi relies only on their core features, which are common in other related technologies. Specifically, we leverage the ability of R-OSGi to convey network failure as application-level exceptions and to reestablish connections once the network becomes available. JBoss AOP effectively modularizes hardening strategies. Although our approach delivers tangible benefits to the distributed component programmers, it also has some inherent limitations.

Advantages DR-OSGi makes it possible to handle network volatility consistently throughout a distributed component application. This means that the most appropriate hardening strategy can be applied to any subset of application components, and the strategies can be switched through a simple change in the configuration file. Furthermore, each strategy is modularized inside a separate OSGi bundle, thus streamlining maintenance and evolution. Finally, modularized strategies can be easily reused across different distributed component applications.

Limitations Creating a pragmatic solution that can be implemented straightforwardly required constraining our design in several respects. For example, we chose to maintain a one-to-one correspondence between application bundles and their hardening strategy bundles. That is, a hardening strategy for all the services in a bundle must be implemented in a single DR-OSGi strategy bundle. Strategy bundle implementations, of course, can combine any hardening strategies. We have made this design choice to simplify the deployment and configuration of strategy bundles. Another limitation is inherited from JBoss AOP, which is loaded by the infrastructure irrespective of whether a hardening strategy will be applied, thus possibly consuming system resources needlessly. This may present an issue in a resource-scarce environment such as an embedded system. A possible solution to this inefficiency would be to extend OSGi with a meta-model that would allow the programmer to systematically extend services.

4 Evaluation

To evaluate the effectiveness and performance properties of DR-OSGi, we have conducted three benchmark experiments and a larger case study.

4.1 Benchmarks

Since R-OSGi can easily distribute any existing OSGi application, our benchmarks use third-party OSGi components accessed remotely across the network.

As our benchmark applications, we have used a remote log service, a remote user administration service, and a distributed search engine.

To create a controlled networking environment with predictable network outage rates, we have used a network emulator—`netem` [2]—to introduce network volatility conditions, including transmission delay, packet loss, packet duplication, and packet re-ordering.

In our experimental setup, we have emulated a network with the round trip time (RTT) metrics equal to 14ms, which is typical for a modern wireless network. To emulate network outages, we used `netem` to generate packets losses at the server. Lossy network conditions were emulated by losing a high number of random packets (i.e., over 30% loss); totally disconnected networks were emulated by losing all the transmitted packets.

The experimental environment has comprised a Fujitsu S7111 laptop (1.8 GHz Intel Dual-Core CPU, 2.5 GB RAM) communicating with a Dell XPS M1330 laptop (2.0 GHz Intel Dual-Core CPU,3 GB RAM) via a IEEE 802.11g wireless LAN, with both laptops running the Sun's client JVM, JDK J2SE 1.6.0_13.

Log Service For this experiment, we used a log service defined by the OSGi specification [19]. The OSGi log service records standard output and error messages printed during a bundle's execution. The service can be configured to log different amounts of messages by calling its `setLevel` methods (the higher the level, the more messages are logged).

Imagine needing to log messages generated by a remote service locally. In this experiment, we have used R-OSGi to access the existing log service of Knopflerfish, a popular, open-source implementation of OSGi. To enable remote access, we have used the surrogation bundle approach to register the existing log service.

Network volatility should not cause a remote log service to stop functioning. Logs are typically examined for a postmortem analysis, for which the actual time when the messages are written to a log file is not important, as long as the messages' timestamps reflect their actual origination time.

In our experiment, we used the log service to record 10 text messages generated consecutively without any delay. The network is available during the remote logging of the first 3 messages. Immediately after logging the third message, the network becomes totally disconnected. Then after the fifth message, the network connection is restored.

We have executed this scenario under two setups: plain R-OSGi and DR-OSGi with a queuing strategy. Recall that queuing works by recording remote service calls when the network is unavailable and replays the recorded calls once the connection is restored. Under the original setup, the remote log service recorded only 8 messages (3 before the disconnection and 5 after). Two messages were lost irretrievably. The hardened version recorded all 10 messages.

Table 1 shows the delay for each message delivery. For the queued messages (columns 4 and 5), the delays is significantly higher than for the other messages. Despite the delay of the queued messages, all the messages are delivered in the order in which they are sent. Since real-time guarantees are not required, we can conclude that the hardening strategy has provided the requisite QoS for the remote log service, allowing it to cope with network volatility.

Table 1. Message delivery delay under a queuing hardening strategy.

Network condition	connection			disconnection		connection				
Message number	1	2	3	4	5	6	7	8	9	10
Sent log time(min:sec)	0:00	1:12	1:21	1:51	2:51	3:19	4:01	4:03	4:42	4:46
Received log time(min:sec)	0:00	1:15	1:21	3:20	3:20	3:20	4:02	4:05	4:42	4:46

User Admin Service For this experiment, we used the *User Admin Service*, which comes as a part of the core OSGi system services. The service authenticates and authorizes users by running their credentials against a database. Oftentimes, this service may need to be accessed remotely. To introduce distribution, we have registered the standard User Admin Service bundle using a surrogation bundle, similar to the approach we took in distributing the log service.

A network outage should not prevent a client from using the User Admin Service, if the client has used the service in the past, and the security policy specifies that user credentials change infrequently and can be cached safely. In other words, the caching hardening strategy must be coordinated with the security policy in place, lest the system's security can be compromised. One way to accomplish this is to avoid caching the authentication data that may change while the network is temporarily unavailable.

We have emulated a scenario in which 100 remote authentication attempts have been made across the network, which randomly suffers disconnections with the rate equal to 1 disconnection per 20 authentication attempts. Disconnections always cause the R-OSGi version of the application to fail. The ability of the DR-OSGi hardened version to continue executing depends on the number of clients. In this simulation, we assume that all the clients use the service equally. Thus, if for example, there were n authentication requests made from m users, then the expected number of authentications performed by a single user is n/m. Since the cache size is set to 5, the hit rate is negatively correlated with the number of users, standing at 100% for 2 and 4, and going down to 90%, 85%, and 78% for 6, 8, and 10 users, respectively.

Distributed Lucene For this experiment, we have used Lucene, a widely-used Java search engine library. Among the capabilities provided by Lucene are indexing files and finding indexes of a given search word. Because searching is computationally intensive, there is great potential benefit in distributing the searching tasks across multiple machines, so that they could be performed in parallel.

Despite several known RMI-based Lucene distributions, for our experiments we have created an R-OSGi distribution, which turned out to be quite straightforward. We have followed a simple Master Worker model, with the Master assigning search tasks to individual Workers as well as collecting and filtering search results. This distribution strategy, depicted in Figure 4, requires that only the Master node be hardened against network volatility. This embarrassingly parallel data distribution arrangement imposes a strict one way communication protocol with the Master always calling Workers but never vice versa.

Once again, a caching hardening strategy has turned out to be most appropriate for hardening the distributed Lucene R-OSGi application. Specifically, every work assignment for individual nodes is used as a key mapped to the returned result. The intuition behind this caching scheme is that files are read-only and searching a file for the same string multiple times must return identical results. For writable files, the caching scheme would have to be modified to invalidate all

the cached results for the changed files. As it turns out, the absolute majority of environments that use Lucene feature read-only files only, including digital books, scientific articles, and news archives.

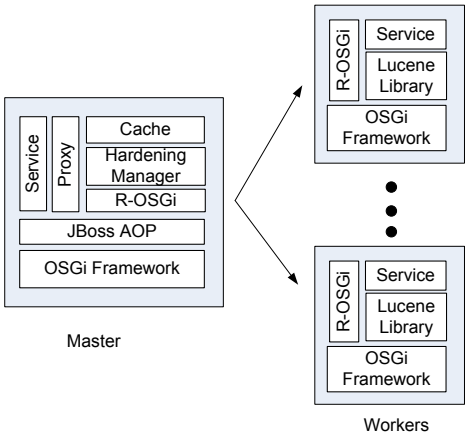

Fig. 4. Distributed Lucene.

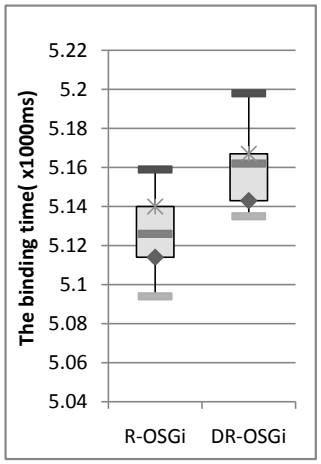

Fig. 5. The Binding Time.

Since distributed Lucene is representative of a large class of realistic applications, we have used it to assess the performance overhead imposed by DR-OSGi. The first benchmark has measured the binding time, which is defined as the total time expended on establishing a remote connection, requesting the service, receiving the interface, and building the remoting proxy. R-OSGi is quite efficient, with R-OSGi application consistently outperforming their RMI versions [22]. The purpose of our benchmark was to ensure that DR-OSGi does not impose an unreasonable performance overhead on top of R-OSGi. As it turns out, there is not a pronounced difference between the binding time of a plain R-OSGi version of Lucene and its hardened with DR-OSGi version, as shown in Figure 5. One could argue that binding is a one-time expense incurred at the very start of a service and as such is not critical.

To distill the pure overhead of DR-OSGi, we have measured the total time it took to synchronously invoke a remote service under three scenarios:

1. running the original R-RSGi version with no network volatility present
2. running the hardened with DR-OSGi version with no network volatility present
3. running the hardened with DR-OSGi version with a randomly introduced complete network disconnection

The measurements are the result of averaging the total time taken by $1 * 10^3$ remote service invocations. To emulate a complete network disconnection, we

have generated a 100% packet loss. While the original R-OSGi version takes 9043.5 ms to execute, the hardened one takes 9321.9 ms, thus incurring only 3% overhead when no volatility is present. When the network becomes unavailable, the DR-OSGi caching strategy improves the performance quite significantly, as it eliminates the need for any computation to be done by the worker node. While, somewhat unrealistically, we used the 100% hit rate to isolate the overhead of DR-OSGi, the actual performance is likely to vary widely depending on the applicaion-specific caching scheme in place.

These performance results indicate that the insignificant performance overhead that DR-OSGi imposes on a hardened distributed application can certainly be justified by the added resiliency to cope with network volatility.

4.2 Case Study

As a larger case study, we have hardened "DNA Hound," a three-tier R-OSGi application for assisting detectives conducting a criminal investigation. The application works by automating the process of analyzing and warehousing DNA evidence, collected at crime scene investigation sites. Figure 6 depicts the architecture of "DNA Hound." The detective collects DNA evidence using a hand-held device, and then sends it to a search facility using a mobile data network (or any other wireless network). The search facility matches the sent DNA evidence against a database of DNA sequences (via parallel processing) and reports if a match is found. The collected DNA evidence is then sent to a crime evidence warehouse for storage.

We have implemented a complete working prototype of "DNA Hound," but in lieu of DNA extracting hardware, we simulated the found DNA evidence by randomly selecting DNA sequences from a GenBank NCBI database [4]. The search is performed using a parallelization of the Smith-Waterman algorithm [24] on a compute cluster.

Hardening "DNA Hound" Because "DNA Hound" is used in the field, it relies on a wireless network that can be unreliable. Therefore, to ensure that the application continues to provide service, we have used DR-OSGi to harden it against network volatility. We have used two hardening strategies implemented as regular OSGi bundles.

Replication. To harden the application for the network volatility that can occur between the hand-held DNA extractor and the analyzer, we have used a replication strategy. Although DNA sequence search is very computationally-intensive, usually requiring parallel processing to shorten the search time, it can also be done sequentially, albeit much slower. With the advance in data storage technologies, even a hand-held device can comfortably store a substantial database of DNA sequences. The DNA search bundle is replicated at the hand-held device. We have used the native OSGi replication facilities to install the search bundle at both sites. When the network is up, the search is performed using a compute cluster at the search site, and the index of the most recently

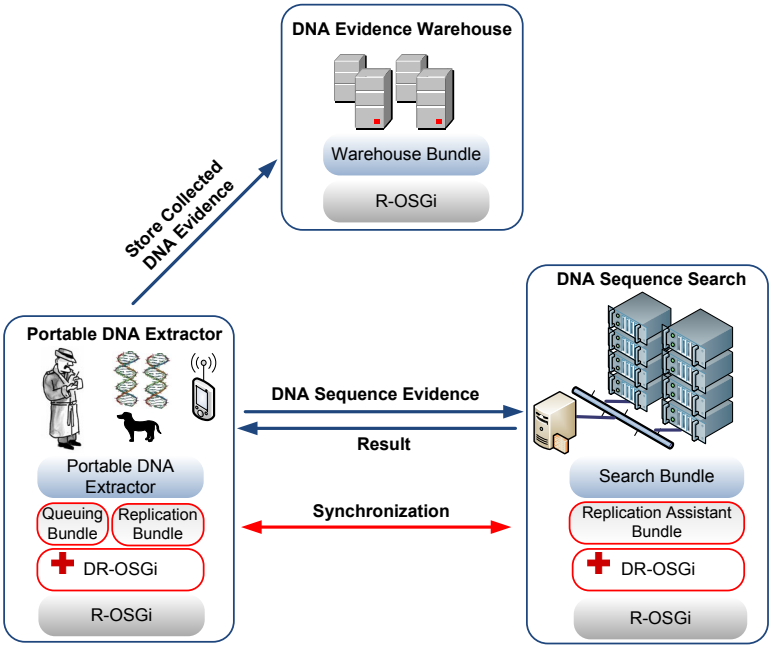

Fig. 6. *DNA Hound* System Architecture.

searched database sequence is periodically sent to the search bundle at the hand-held site. Once the network becomes unavailable, the search bundle at the hand-held site continues the search locally, using an inefficient sequential algorithm; the search is continued from the index of the last searched sequence at the cluster. If the index is not up-to-date, then some overlap in the search will occur. Once the connection goes back up, the cluster could then report any matches found while the network was not available.

Queuing. To harden the application for the network volatility that can occur between the hand-held DNA extractor and the criminal evidence warehouse, we have used a queuing strategy. The calls to store a new piece of DNA evidence are queued up at the hand-held site once the network becomes unavailable. Then the queued calls are resent to the warehouse once the network connection is restored.

Discussion The original R-OSGi version of the application was written without any functionality enabling it to cope with network volatility–it thus fails immediately once either network link is lost. DR-OSGi made it possible to harden this unaware application, so that it can meaningfully continue its operation in the presence of network volatility, thus improving the application's utility and safety. This demonstrates how DR-OSGi makes it possible to treat network volatility resiliency as a separate concern that can be implemented separately and added to an existing application. Furthermore, the queuing bundle came from the library of standard hardening strategy bundles that are part of our DR-OSGi

distribution, thus requiring no programmer effort. The replication bundle was custom tailored for this application, but we are currently working on generalizing the implementation, so that only the synchronization functionality would require custom coding.

5 Related Work

DR-OSGi derives its hardening strategies from a recent survey of disconnected operation techniques by Mikic-Rakic and Medvidovic [15]. These techniques are used by several systems, including the Rover toolkit [11], Mobile Extension [5], Odyssey [16], and FarGo-DA [27]. Unlike these systems, DR-OSGi enables the programmer to harden distributed applications without having to modify their source code explicitly. By avoiding ad-hoc modification that can be tedious and error-prone, DR-OSGi not only hardens applications more systematically, but also enables greater reuse of the hardening strategies across different applications.

Aldrich et al.'s ArchJava [3] extends Java to integrate architectural specifications with the implementation by providing language support for user-defined connectors. Their techniques bears similarity to DR-OSGi in separating reusable connection logic from the application logic and integrating them together systematically. ArchJava, however, operates at the source code level, using its language extension to express different connectors. DR-OSGi is a middleware solution that does not need to modify the source code.

Sadjadi and McKinley's adaptive CORBA template (ACT) enables CORBA applications to adapt to unanticipated changes [23]. To do so, ACT employs a generic interceptor, a type of CORBA portable request interceptor [17] that works around the constraints of replying to intercepted requests or modifying the invoked method's parameters. Specifically, a generic interceptor forwards requests to a proxy, a CORBA object that can reply and modify the requests. Similarly to DR-OSGi, ACT introduces additional functionality to a distributed application without modifying its code explicitly. Using ACT to harden against network volatility, however, would require that portable interceptors be available, which may not be the case for many distributed component infrastructures including R-OSGi.

A number of techniques for making existing systems fault tolerant [7, 20, 26] are related to our approach. GRAFT [26] automatically specializes middleware for fault-tolerance. It employs the Component Availability Modeling Language (CAML) to annotate a distributed application's model, and then automatically specializes the application's middleware for domain-specific fault-tolerant requirements. While GRAFT requires that the programmer express the requested fault-tolerance functionality at the model level using a domain-specific language, DR-OSGi provides a simple Java API for implementing hardening strategies as OSGi bundles, which it then manages at runtime.

Our idea of hardening against network volatility was inspired by *security hardening*, a systematic approach to making a pre-existing program artifact more

secure such as Wuyts et al's recent work [28]. Our approach hardens distributed components to become more resilient against network volatility.

6 Future Work and Conclusions

One future work direction will assess the generality of our approach by applying it to other distributed component infrastructures. Another direction will focus on identifying suitable hardening strategies through the program analysis of distributed components.

We have presented DR-OSGi, a promising approach for systematically hardening distributed components to cope with network volatility. The reference implementation features an extensible framework for deploying hardening strategies, with caching, queuing, and replication used to demonstrate the effectiveness of our approach. As we rely on greater numbers of network-enabled devices with network volatility remaining a permanent presence, the importance of hardening distributed components will only increase, motivating the creation of systematic and flexible hardening approaches as showcased by DR-OSGi.

Availability: DR-OSGi and all the applications described in the paper can be downloaded from `http://research.cs.vt.edu/vtspaces/drosgi`.

References

1. CCA-Forum. `http://www.cca-forum.org/`.
2. Net:Netem. `http://www.linuxfoundation.org/en/Net:Netem/`.
3. J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'03)*, July 2003.
4. D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler. Genbank. *Nucleic Acids Res.*, 30:17–20, 2002.
5. M. Dahlin, B. Chandra, L. Gao, A. Khoja, A. Nayate, A. Razzaq, and A. Sewani. Using mobile extensions to support disconnected services. Technical Report CS-TR-00-20, University of Texas at Austin, 2000.
6. C. L. Fullmer and J. Garcia-Luna-Aceves. Solutions to hidden terminal problems in wireless networks. In *In Proceedings ACM SIGCOMM*, pages 39–49, 1997.
7. J. L. Herrero, F. Sanchez, O. Sanchez, and M. Toro. Fault tolerance AOP approach. In *Workshop on AOP and Separation of Concerns*, pages 44–52, 2001.
8. B. Hull, K. Jamieson, and H. Balakrishnan. Mitigating congestion in wireless sensor networks. In *ACM SenSys 2004*, Baltimore, MD, November 2004.
9. K. Jain, J. Padhye, V. N. Padmanabhan, and L. Qiu. Impact of interference on multi-hop wireless network performance. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 66–80, New York, NY, USA, 2003. ACM.
10. JBoss AOP. `http://www.jboss.org/jbossaop`.
11. A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. . Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

12. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, London, UK, 2001. Springer-Verlag.
13. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming(ECOOP 97)*, pages 220–242, 1997.
14. Microsoft. Component Object Model (COM).
15. M. Mikic-Rakic and N. Medvidovic. A classification of disconnected operation techniques. In *Proceedings of the 32nd EUROMICRO Conference on Software engineering and Advanced Applications (EUROMICRO-SEAA'06)*, 2006.
16. B. D. Noble, M. Sayanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.
17. Object Management Group. The common object request broker: Architecture and specification version 3.0. http://doc.ece.uci.edu/CORBA/formal/02-06-33.pdf, July 2003.
18. Object Management Group. The CORBA component model specification. Specification, Object Management Group, 2006.
19. OSGi Alliance. OSGi release 4.1 specification. Specification, 2007.
20. A. Polze, J. Schwarz, and M. Malek. Automatic generation of fault-tolerant CORBA-services. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 2000)*, 2000.
21. J. S. Rellermeyer and G. Alonso. Concierge: a service platform for resource-constrained devices. In *the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 245 – 258, 2007.
22. J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed applications through software modularization. In *Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference (Middleware 2007)*, November 2007.
23. S. M. Sadjadi and P. K. McKinley. ACT: An adaptive CORBA template to support unanticipated adaptation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 74 – 83, 2004.
24. T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.
25. Spring Framework. `http://www.springsource.org/`.
26. S. Tambe, A. Dabholkar, J. Balasubramanian, and A. Gokhale. Automating middleware specializations for fault tolerance. In *Proceedings of the International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, March 2009.
27. Y. Weinsberg and I. Ben-Shaul. A programming model and system support for disconnected-aware applications on resource-constrained devices. In *Proceedings of the 24th International Conference on Software Engineering*, pages 374 – 384, Orlando, Florida, May 2002.
28. K. Wuyts, R. Scandariato, G. Claeys, and W. Joosen. Hardening XDS-based architectures. In *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, pages 18–25, Washington, DC, USA, 2008. IEEE Computer Society.
29. M. Zhang and R. Wolff. Crossing the digital divide: cost-effective broadband wireless access for rural and remote areas. *Communications Magazine, IEEE*, 42(2):99–105, Feb 2004.