The Contingency Management Framework

Version 1.0

Jakob Bardram

IT University Technical Report Series **TR-2009-121**

ISSN 1600–6100 **December 2009**

ISSN 1600–6100

ISBN 9788779492035

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: **+45 72 18 50 00**
Telefax: **+45 72 18 50 01**
Web **www.itu.dk**

Contents

List of Figures

Document Management

Version	Date	Init.	Description
1.0	18/3-2009	JB	Initial version
1.1	5/5-2009	JB	Details on programming added to section 4.
1.2	20/12-2009	JB	Final proff-reading and release as ITU Technical Report.

Chapter 1

Introduction

This technical report describes version 1.0 of the Contingency Management Framework – *CMF* . *CMF* is designed to support contingency management in Ubiquitous Computing (Ubicomp) systems design. The purpose of this document is to describe the technical design and implementation of version 1.0.

This document describes the overall goal of *CMF* (section 1.1) and its environmental assumptions (section 1.2); it provides an overview of the key concepts in *CMF* (chapter 2); it documents the overall *CMF* runtime infrastructure and architecture (chapter 3); and it provides an overview of the *CMF* programming model, including special programming language notation for contingency management assertions (chapter 4).

1.1 Goals of the *CMF* Platform

The concept of contingency management is related to exception handling, fault-tolerant, and resource-aware systems. Exception handling is a language construction for handling exceptional states in the program flow. Typically, however, there is little to do once an exception is thrown, rendering the state of the executing program unpredictable, for example in the case of a network error. Contingency management, on the other hand, seeks to monitor vital resource (e.g. the network) and ensures that appropriate coping strategies are applied *before* the program starts to use a resource, which may result in an exception. For example, avoid contacting the network in the case of network unavailability. Contingency management is central to research within ubiquitous computing, since it addresses reliable computing in heterogeneous, volatile execution environments while utilizing concepts and technologies for context-aware computing, resource discovery, and resource-aware computing.

To enumerate the goals:

- Pro-active exception handling in a volatile runtime environment
- Platform support for resource-aware and context-aware computing
- Transaction support
- Language support
- API support
- Separation of contingency management mechanisms and runtime/platform implementation

To our knowledge, this approach of moving from post-hoc exception handling to anticipatory (or pro-active) contingency management is new. Some initial thoughts on contingency management have been discussed as part of the European PalCom project [1].

1.2 Environmental Assumptions

CMF version 1.0 is implemented in Java and used the Java Development Kit (JDK), Standard Edition (SE)[1]. It has been implemented and tested on Java SE 5.

[1] http://java.sun.com/javase/

Chapter 2

Key Concepts

2.1 Contingency Management

An UML class diagram of the core concepts/classes in the contingency management part of *CMF* is shown in figure 2.1.

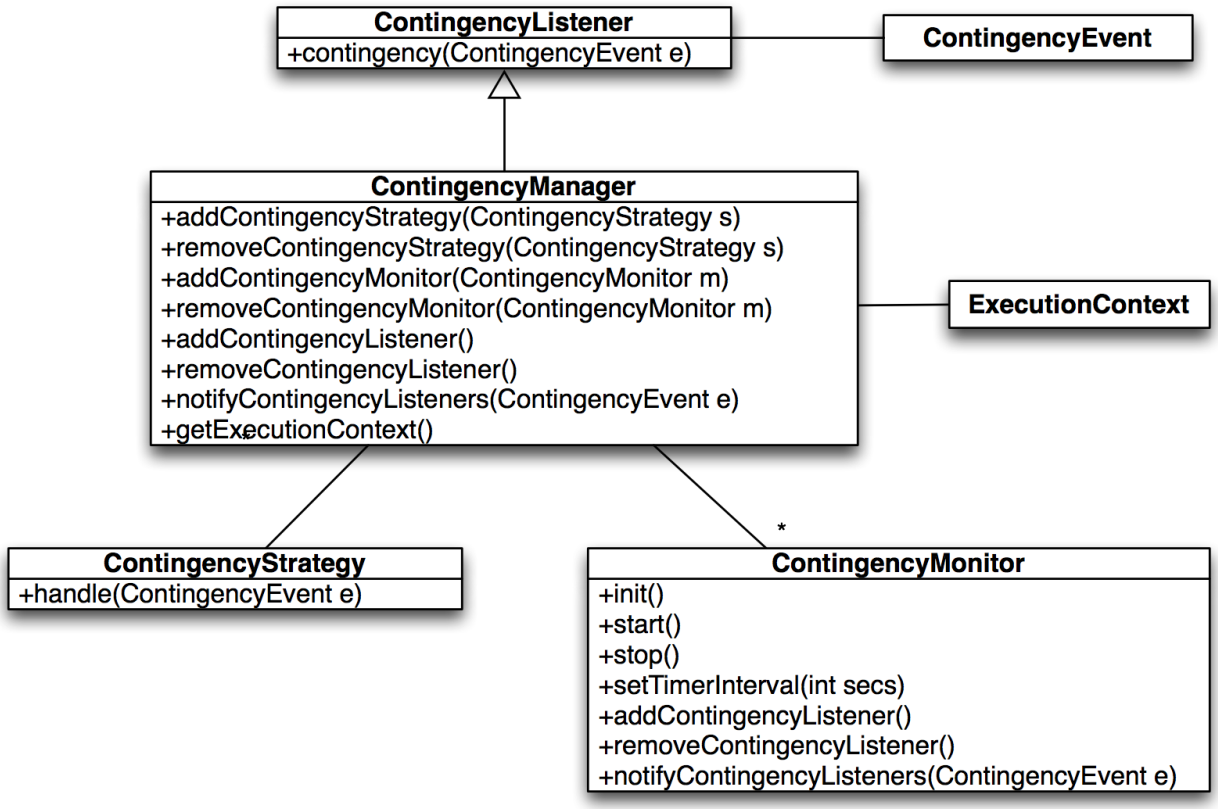

Figure 2.1: UML class diagram for the contingency management sub-system of *CMF*

The core process is the *Contingency Manager* which is deployed on each executing node / device. A contingency manager is responsible for analyzing the current state of relevant resources (see section 2.2) and for handling any contingencies which may arise during runtime. Analysis of the current state of affairs is done through a set of *Con-*

tingency Monitors. A contingency monitor monitors both OS-level events concerning e.g. network, CPU load, and memory usage, as well as events related to the user's context, like location, status, or activity. *Contingency Listeners* can subscribe to being notified about *Contingency Events* discovered by a monitor. Note that a contingency manager is also a contingency listener and the manager hence listen for contingency events discovered by its monitors. This also implies that contingency managers can cooperate and listen for contingency events occurring in each other (more on this in chapter 3).

2.1.1 Contingency Manager

A contingency manager is responsible for analyzing the current state of relevant resources (see section 2.2) and for handling any contingencies which may arise during runtime. `ContingencyManager` is a Java interface and can be implemented either as part of a client program, or it may run as a separate process. *CMF* contains a default implementation called `GenericContingencyManager` which can either run as a separate process or be embedded into an existing application.

The contingency manager analyses the current state of its environment through contingency monitors, which can be added and removed from the manager using the `addContextMonitor()` and `removeContextMonitor()` methods. Contingency monitors reports back to the contingency manager through the `contingency` method specifying a `ContingencyEvent`. Examples of contingency events are changes in CPU load or network connectivity.

The contingency manager handles contingencies through the use of contingency strategies. Contingency strategies can be added and removed from the manager using the `addContingencyStrategy()` and `removeContingencyStrategy()` methods. When adding a strategy, a class type of type `ContingencyEvent` is specified. This type specify which types of contingency events this specific strategy applies for. For example, a strategy for handling network failure can be added to the the contingency manager and the type parameter specify a contingency event type related to network connectivity. Once a network monitor raises a contingency event of type network connectivity (including sub-types), the network strategy is notified using the `handle` method.

When a contingency manager's `contingency` event is raised, the manager looks up relevant strategies and apply them in a random order. This dynamic behavior is illustrated in the interaction diagram in figure 2.2 for only one monitor and stategy. Note, however, that there can be multiple monitors and strategies associated with a contingency manager.

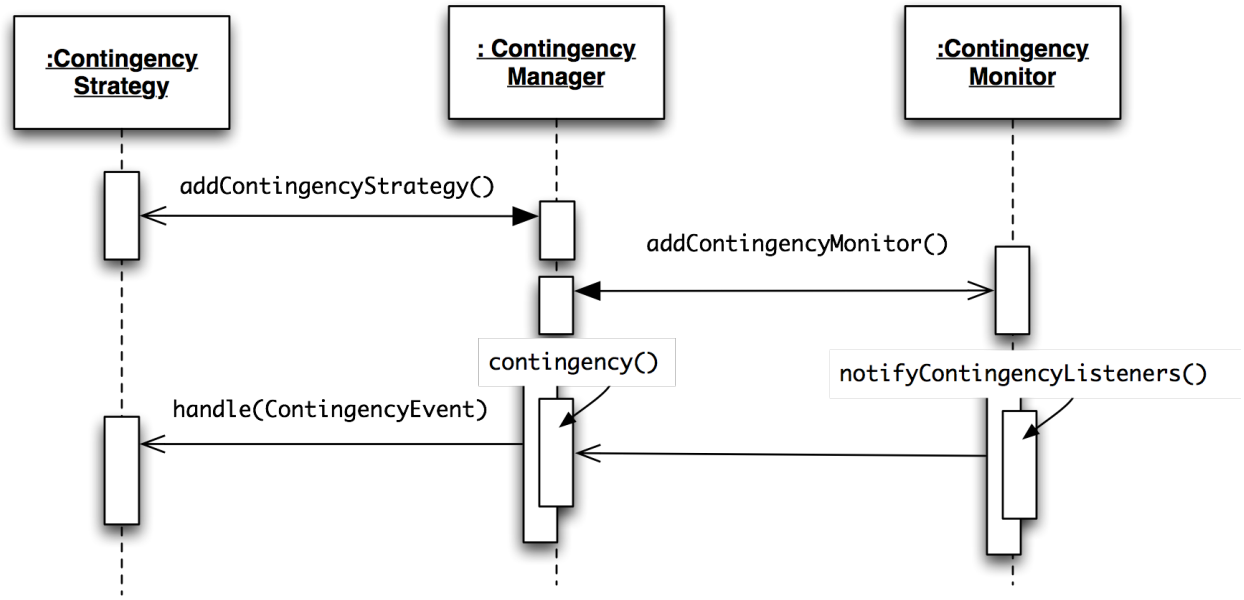

Figure 2.2: UML interaction diagram for the contingency management including setting up contingency monitors and stategies.

Finally, the contingency manager provides access to the current execution context of the manager's host device. This is modeled by the `ExecutionContext` class, which ties into the resource- and context-aware subpart of *CMF* (see section 2.2).

2.1.2 Contingency Listener

The `ContingencyListener` is a simple interface that anyone interested in listening in on contingency events may implement. It contains only one method, namely the `contingency(ContingencyEvent event)` method, which is called when an contingency event occurs. Contingency listeners add themselves to contingency monitors using the `addContingencyListener()` method. Note that the contingency manager itself is a contingency listener on all its associated monitors and that a resource (see section 2.2) is an contingency listener as well.

2.1.3 Contingency Monitor

The `ContingencyMonitor` is a Java interface specifying a contingency monitor, which monitors specific contingent aspects of the host device. Examples of contingency monitors are network monitors, CPU monitors, and I/O monitors. The class `AbstractContingencyMonitor` is an abstract implementation of a monitor, which can be used as a superclass for the design of more specific monitors.

A contingency monitor has methods for adding and removing contingency listeners, and has the method `notifyContingencyLis` `event)`, which notifies all contingency listeners currently listening to this monitor. In addition, a contingency monitor has a set of life-cycle methods for initializing, starting, and stopping the monitor. A monitor runs in its own thread and reports back to its listeners on a regular basis specified in seconds in the `setTimerInterval(int secs)` method.

2.1.4 Contingency Strategy

The manager handles contingencies through different *Contingency Strategies*, which enlist to handle different types of *Contingency Events*. The `ContingencyStrategy` is a Java interface specifying a contingency strategy, which is used to implement a strategy for handling contingency events. It contains only one method, namely the `handle(ContingencyEvent` `event)` method, which is called when an contingency event occurs that this strategy needs to handle.

Contingency strategies are added to a contingency manager using the `addContingencyStrategy()` method. The `type` argument specify which type of contingency events that this strategy handles. Several strategies can be added to handle the same event type, and the same strategy may handle several event type. For example, one or more strategies can enlist to act when the network comes and goes, and other strategies can subscribe to handling user/device mobility.

Contingency strategies are often domain specific and tied closely to the application, and the actual implementation of the strategies will often take place in different application-specific components. For example, in the Mini-Grid Framework [1], the components responsible for peer discovery and task submission, scheduling, and distribution implements different contingency strategies which handle different types of contingency events. A simple example is not to try to distribute jobs while disconnected. A more complex example is to try to re-distribute running jobs before a laptop is suspended or is about to leave the biology laboratory.

2.2 Resource and Context Awareness

An UML class diagram of the core concepts/classes in the resource and context awareness part of *CMF* is shown in figure 2.3.

The contingency manager keeps track of its runtime environment by storing *Resouce* state information as part of the *Execution Context*. The resource context is generally accessible to the rest of the infrastructure and can hence be used to coordinate resource-aware computing. For example, if one monitor discover that the network has disappeared,

[1] http://www.itu.dk/research/mini-grid/

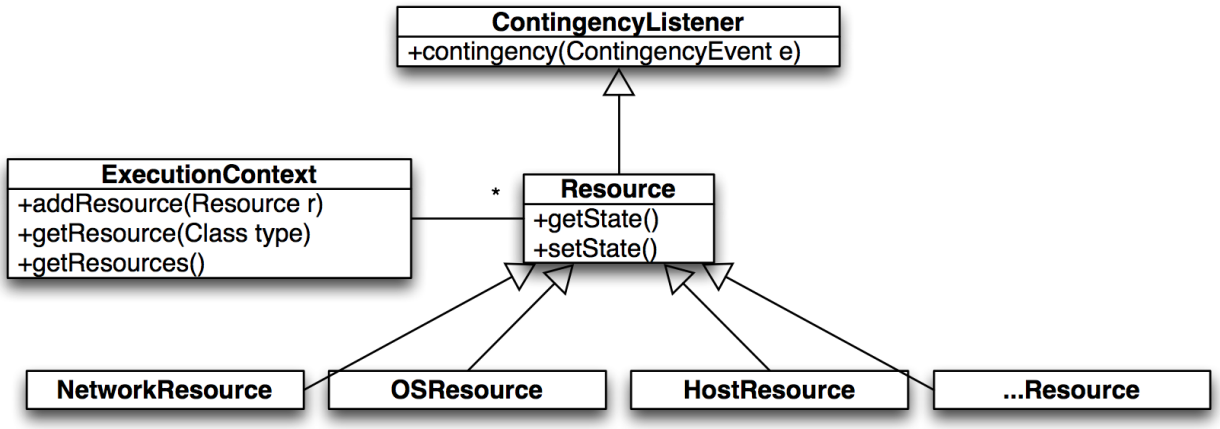

Figure 2.3: UML class diagram for the resource and context awareness sub-system of *CMF*

this knowledge will be available to the rest of the system, and the system would then pro-actively avoid using the network. This design pattern resembles the Servlet Context in the J2EE architecture.

2.2.1 Execution Context

The `ExecutionContext` is a Java interface intended to hold a set of resource. The class `GenericExecutionContext` provides a generic implementation that natively handles host, network, operation system, and display resources. The execution context can be trusted to always give an up-to-date state of the local execution environment.

In this manner, the execution context maintains system awareness about the execution context, which allow for resource-aware design of applications. We shall return to how this can be implemented in chapter 4.1.

2.2.2 Resource

The `Resource` is a Java interface intended to model a resource on the local device. Several implementation classes exists in *CMF* , including `NetworkResource` that provides state information on the local network connectivity and bandwidth; `OSResource` that provides state information about the local operating system; and `HostResource` which provides state information on the local host, including what type of device it is (laptop, desktop, etc.).

This model is primarily targeted at providing resource-aware computing. However, it can also be extended to support context-aware computing in a simplified way. For example, if the location of a portable device is important to monitor and handle in an application, a location monitor and a location contingency event can be implemented. The host resource may then listen in on location events and in this way keep track of the host's location.

Note that a resource is also a contingency listener, which can subscribe to listen into contingency events from contingency monitors. Hence, a typical setup is that the network resource adds itself as a listener to the network monitor, and thereby being informed about changes in the network state, which then is saved as state information in the resource.

2.3 Distributed Transactions Support

An UML class diagram of the core concepts/classes in the distributed transaction part of *CMF* is shown in figure 2.4.

Even though distributed transaction is not part of monitoring and handling contingencies, it is still central to contingency management in order to maintain consistency in contingent situations. In situations where a contingency (or an error) occurs, transaction mechanisms are central to roll back to a consistent state. For example, if one node in the Mini-Grid infrastructure starts distributing tasks to other nodes, and one of these nodes fail, it is essential that the scheduling node can roll back to a previous state before distribution of tasks started.

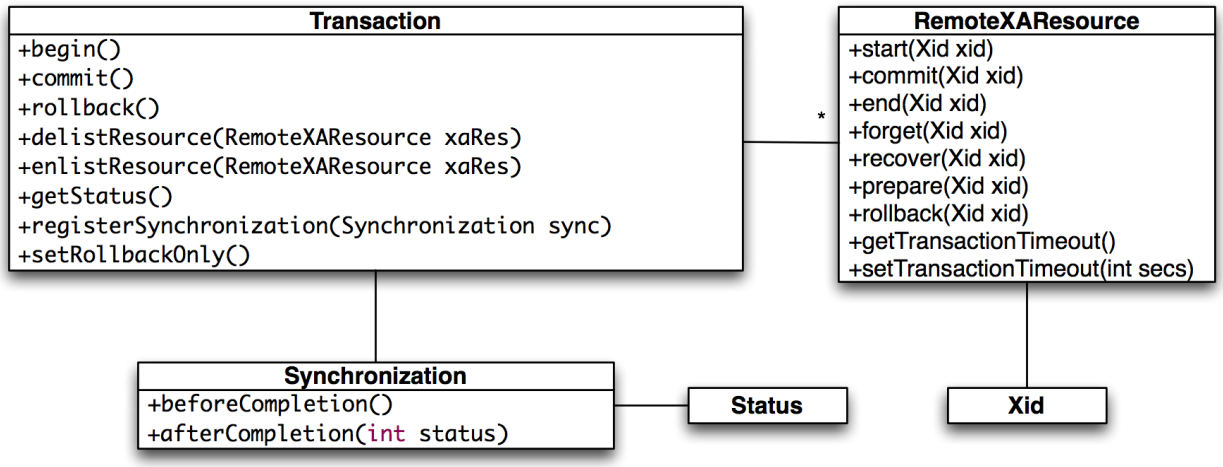

Figure 2.4: UML class diagram for the distributed transaction sub-system of *CMF*

It should be noted that the *CMF* support for distributed tranactions are modeled closely to the Java Transaction API (JTA)[2].

2.3.1 Transaction and Xid

The `Transaction` interface allows operations to be performed against the transaction in the target Transaction object. A Transaction object is created corresponding to each global transaction creation. The Transaction object can be used for resource enlistment, synchronization registration, transaction completion, and status query operations. `Xid` is a global unique transaction identifier. Transaction commit follows a classic two-phase commit protocol. This is illustrated in figure 2.5.

2.3.2 RemoteXAResource

The `RemoteXAResource` interface defines the contract between a Resource Manager and a Transaction in a distributed transaction processing (DTP) environment. For example, in 'classic' J2EE programming, a JDBC driver or a JMS provider implements this interface to support the association between a global transaction and a database or message service connection.

The RemoteXAResource interface can be supported by any transactional resource that is intended to be used by application programs in an environment where transactions can be controlled by some transaction manager. An example of such a resource is a database management system. An application may access data through multiple database connections. Each database connection is enlisted with the transaction manager as a transactional resource. The transaction manager obtains an RemoteXAResource for each connection participating in a global transaction. The transaction uses the `begin()` method to associate the global transaction with the resource, and it uses the end method to disassociate the transaction from the resource. The resource manager is responsible for associating the global transaction to all work performed on its data between the start and end method invocations.

At transaction commit time, the resource managers are informed by the transaction to prepare, commit, or rollback a transaction according to the two-phase commit protocol.

2.3.3 Synchronization and Status

The transaction supports a synchronization mechanism that allows interested parties to be notified before and after the transaction completes. Using the `registerSynchronization` method, an application can register a

[2]http://java.sun.com/jta

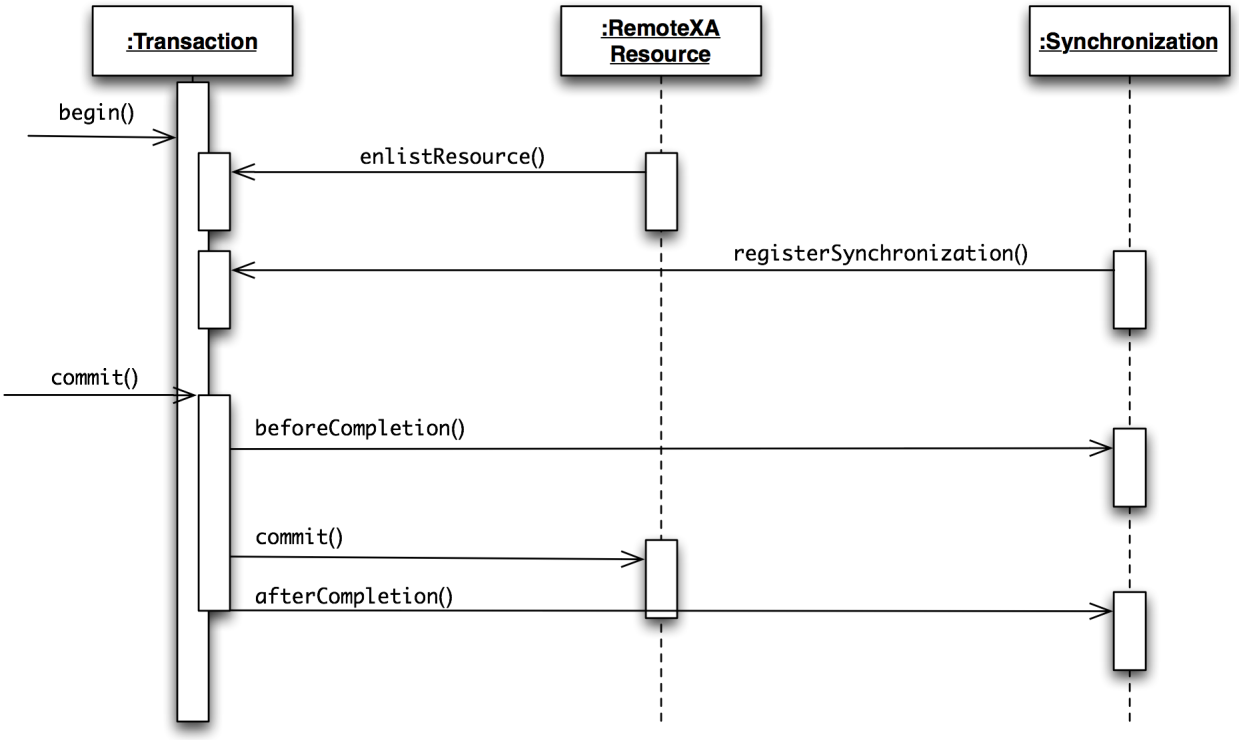

Figure 2.5: UML interaction diagram for resource enlistment, synchronization, and transaction completion (commit and rollback).

`Synchronization` object for the transaction. The transction invokes the `beforeCompletion()` method prior to starting the two-phase transaction commit process. After the transaction is completed, the transaction invokes the `afterCompletion(int status)` method.

`Status` specify the different status that a transaction may have, like `STATUS_COMMITTED` or `STATUS_ROLLEDBACK` .

Chapter 3

The *CMF* Architecture

An UML deployment diagram of the core classes and components in *CMF* is shown in figure 3.1.

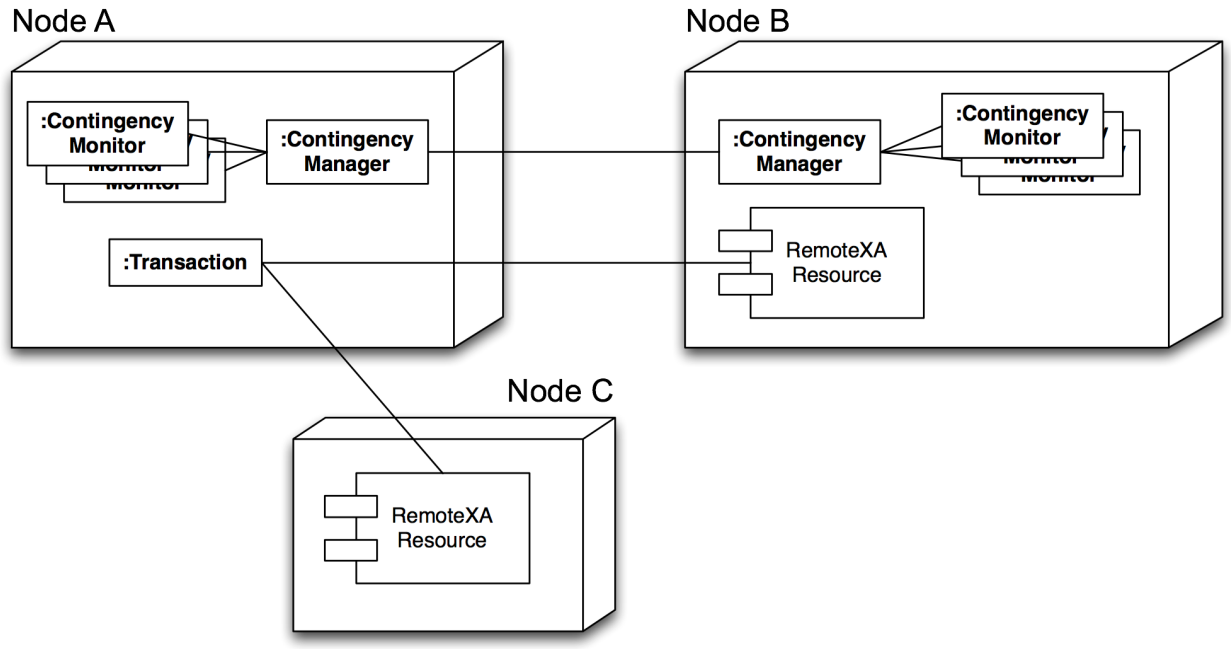

Figure 3.1: UML deployment diagram for *CMF*

Each node runs one or more contingency managers, which may listen to each other – also remotely. Hence, the contingency manager on node A listen to contingency event on node B. The contingency monitors reports to their local contingency manager. A set of remote transaction resources may be enlisted to a transaction. The transaction on node A has remote resources at node B and C.

Chapter 4

The *CMF* Progamming Model

CMF can be used in two ways:

1. as a framework using the Java interfaces and classes API
2. using a special-purpose syntax for contingency-aware programming

4.1 The *CMF* Java Application Programmer Interface

Using the *CMF* Java API primarily implies implementing and extending the Java interface and classes of the *CMF* framework. Important interfaces and abstract/generic classes to implement, extend, and use includes:

1. `GenericContingencyManager` – provides a default implementation of a contingency manager.
2. `AbstractContingencyMonitor` – provides a simple, abstract implementation of a monitor. Runs as a Timer Task, and the programmer needs to implement the logic of the monitor in the `run()` method.
3. `ContingencyStrategy` – the main interface to implement for a contingency strategy. Need to implement the `handle` method.
4. `GenericExecutionContext` – a general implementation of an execution context which can hold a range of resources. This implementation supports handling `HostResource`, `NetworkResource`, `DisplayResource`, and `OSResource`.
5. `AbstractResource` – a simple abstract implementation of a resource.
6. `SimpleNetworkMonitor` – a simple example of a contingency monitor targeted at network monitoring.
7. `NetworkContingencyEvent` – a simple example of a contingency event used by the network monitor.

As a simple example, let us consider how to support contingency management in the Mini-Grid framework. Figure 4.1 shows the UML class diagram for the simplified version of the mini-grid framework. The main classes includes `MiniGrid` as the main application, which holds references to a `ContingencyManager` and a `TaskScheduler`. The `TaskScheduler` is responsible for scheduling the execution of a `BagOfTask` consisting of a list of `Task` objects. Depending on the status of the local machine, the task scheduler can apply different scheduling strategies. In this example, depending on whether the host machine is online or not, the task scheduler either apply a local or distributed task scheduling strategy, implemented in the `LocalTaksSchedulingStrategy` and the `DistributedTaskSchedulingStrategy` classes respectively. The network status is constantly monitored by the `SimpleNetworkMonitor`. By implementing a `NetworkStrategy`, the task scheduler listen to network contingency events, and update is local `TaksScheduling Strategy` accordingly. A similar mechanisms is used

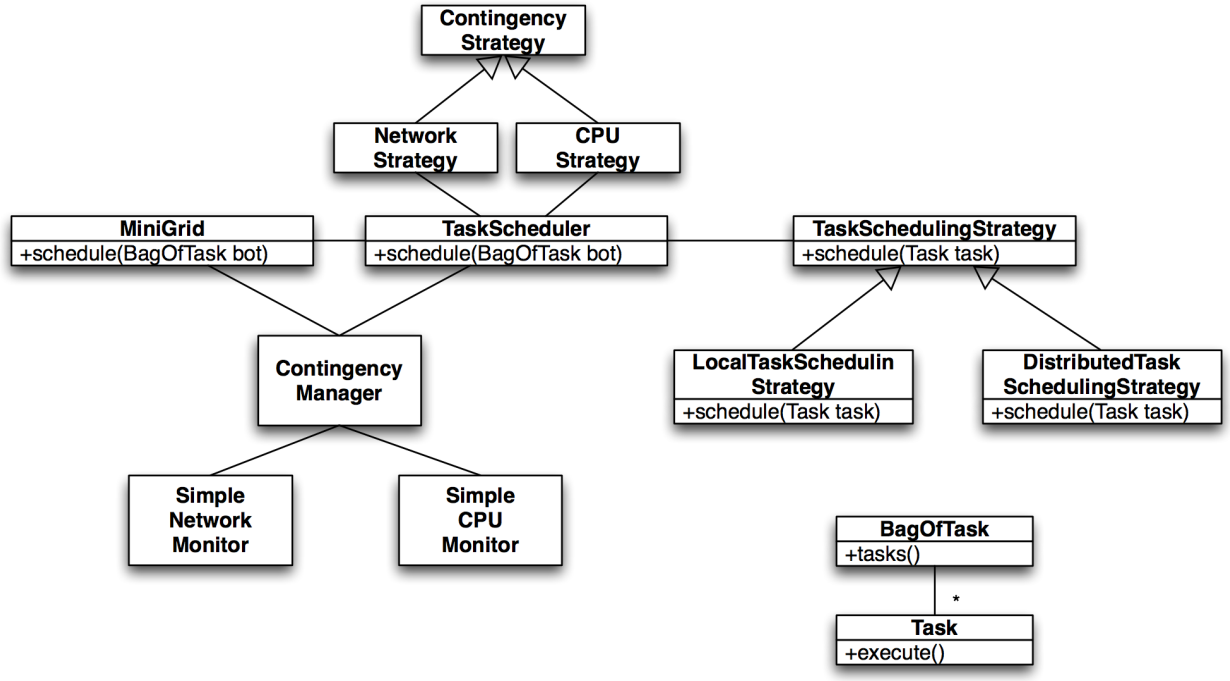

Figure 4.1: UML deployment diagram for *CMF*

```
public class MiniGrid {
   private ContingencyManager mgr = null;
   private TaskScheduler scheduler = null;
   private ContingencyMonitor network = null;
   private ContingencyMonitor cpu = null;
   private NetworkResource nr = null;

   public MiniGrid() {
      this.init();
   }

   private void init() {
      getContingencyManager().addContingencyMonitor(getNetworkMonitor(), getNetworkResource());
      getContingencyManager().addContingencyMonitor(getCPUMonitor(), null);

      getTaskScheduler();
   }

   ...
}
```

Figure 4.2: The Mini-Grid Framework main application code

by monitoring local CPU load; if the local CPU is too busy, the the remote scheduling strategy takes preference over the local one, while taking into account the online status.

Below are some code excerpt illustrating how the *CMF* API is used in this Mini-Grid Framework. Figure 4.2 shows that the mini-grid application holds references to a contingency manager, a task scheduler, two contingency monitors (for network and CPU respectively), and a network resource. In the `init()` method, the contingency manager is initialized, the two monitors are added to it, and the task scheduler is initialized using the `getTaskScheduler()` method.

```
public class TaskScheduler {

   ContingencyManager _mgr = null;
   TaskSchedulingStrategy strategy = null;

   public TaskScheduler(ContingencyManager mgr) {
      this._mgr = mgr;
      this._mgr.addContingencyStrategy(NetworkContingencyEvent.class, new NetworkStrategy());
      this._mgr.addContingencyStrategy(CPUContingencyEvent.class, new CPUStrategy());
   }
   ...
}
```

Figure 4.3: The Task Scheduler code

Figure 4.3 illustrated the initialization of the `TaskScheduler`. In the constructor, two contingency strategies are added to the context manager handling network and CPU contingency events respectively. Figure 4.4 shows the `NetworkStrategy` class, which illustrates how the task scheduling strategy of the task scheduler is changed to use the `LocalTaskSchedulingStrategy` if the device is offline, and to use the `DistrubutedTaskSchedulingStrategy` if the device is online. Similarly, Figure 4.5 shows the `CPUStrategy` class, which illustrates how the task scheduling strategy of the task scheduler is changed to use the `DistrubutedTaskScheduling Strategy` if the device is online and the cpu utilization is greater than 10%, and the `LocalTaskSchedulingStrategy` otherwise

```
private class NetworkStrategy implements ContingencyStrategy {

   public NetworkStrategy() { }

   public void handle(ContingencyEvent event) {
      NetworkContingencyEvent nwce = (NetworkContingencyEvent) event;
      if (nwce.isOnline()) {
         strategy = new DistributedTaskSchedulingStrategy();
      }
      else {
         strategy = new LocalTaskSchedulingStrategy();
      }
   }
}
```

Figure 4.4: The Network Strategy code

The code in figure 4.5 illustrates two ways of using the *CMF* framework API. One way is to listen to contingency events like changes to the CPU utilization and then react to this event according to some strategy or application logic. The other way is to ask the contingency manager for the state of a critical resource before using it, like when the CPU strategy asks about the state of the network, and ensure that the node is online before using the distributed scheduling strategy.

Figure 4.6 shown a simple case of using the contingency manager to ask about status on a critical resource (the network) before trying to use it. In this way, the likelihood of raising an exception is reduced, and the programmer has the option to move from passive handling of exceptions, to a more pro-active mode of programming where resources and their state is investigated before accessed.

4.2 The *CMF* Language Support

The code in Figure 4.6 is not entirely optimal, since IOExceptions may occur during the download. Hence, the creation of the socket and the while loop ought to be surrounded with a try-catch clause. In order to write such statements in a more appropriate way, we suggest a small extension to the Java language; the `try` statement should include a boolean

```
private class CPUStrategy implements ContingencyStrategy {

   public CPUStrategy() {}

   public void handle(ContingencyEvent event) {
      CPUContingencyEvent cpuce = (CPUContingencyEvent) event;
      int online = getContingencyManager().getExecutionContext().
          getResource(NetworkResource.class).getState();
      if ((online != NetworkResource.NET_NONE) && (cpuce.getLoad()) > 10) {
         strategy = new DistributedTaskSchedulingStrategy();
      }
      else {
         strategy = new LocalTaskSchedulingStrategy();
      }
   }
}
```

Figure 4.5: The CPU Strategy code

```
if (getContingencyManager().getExecutionContext().
      getResource(NetworkResource.class).getState() == NetworkResource.NET_NONE) {
   // No connectivity
   System.out.println("No connectivity");
} else {
   //Connectivity, open socket and start downloading content.
   socket = new Socket(ip_address, port);
   while (socket.getInputStream().read() != -1) {
      ...
   }
}
```

Figure 4.6: The use of the *CMF* framework for pro-active programming

condition which has to be true in order for the try-catch block to be executed. The syntax would be as shown in Figure 4.7.

```
try (<expression>) {
   ...
} catch (<exception>) {
   ...
}
```

Figure 4.7: Syntax for the contingency-management `try` statement.

The example from Figure 4.6 could then be rewritten like shown in Figure 4.8. In this case the try clause contain a boolean expression. If this expression evaluates to true, the the try block is executed – otherwise not. This is a simple syntactical extension of Java for more pro-active error and contingency management.

```
boolean online = getContingencyManager().getExecutionContext().
             getResource(NetworkResource.class).getState();
try (online > NetworkResource.NET_NONE) {
   //Connectivity, open socket and start downloading content.
   socket = new Socket(ip_address, port);
   while (socket.getInputStream().read() != -1) {
      ...
   }
} catch (IOException ioe) {
   // No connectivity
   System.out.println("No connectivity");
}
```

Figure 4.8: Example of using the `try` statement for pro-active programming.

Bibliography

[1] Jakob E. Bardram and Ulrik P. Schultz. Contingency Management - Palcom Working Note #30. Technical report, Palcom Project IST-002057, 2004.