International Journal of Distributed Sensor Networks
Manuscript Submission

FlexFT: A Generic Framework for Developing Fault-tolerant Applications in the Sensor Web

Delano Medeiros Beder · Jó Ueyama · João Porto de Albuquerque · Marcos Lordello Chaim

Received: date / Accepted: date

Abstract Fault-tolerant systems are expected to operate in a variety of devices ranging from standard PCs to embedded devices. In addition, the emergence of new software technologies has required these applications to meet the needs of heterogeneous software platforms. However, the existing approaches to build fault-tolerant systems are often targeted at a particular platform and software technology. The objective of this paper is to discuss the use of FlexFT — a generic component-based framework for the construction of adaptive fault tolerant systems that can integrate and re-use technologies and deploy them across heterogeneous devices. Furthermore, FlexFT provides a standardized and interoperable interface for sensor observations by relying upon the "Sensor Web" paradigm established by the Open Geospatial Consortium (OGC). We have implemented a Java prototype of our framework and evaluated the potential benefits by carrying out case-studies and performance measurements. By implementing and deploying these case studies in standard PCs as well as in sensor nodes, we show that FlexFT can cope with the problem of a wide degree of heterogeneity with minimal resource overheads.

Keywords Fault Tolerance, Middleware, Sensor Web

Delano Medeiros Beder
DC, Federal University of São Carlos (UFSCar)
São Carlos, 13565-905, Brazil
E-mail: delano@dc.ufscar.br

Jó Ueyama & João Porto de Albuquerque
ICMC, University of São Paulo(USP)
São Carlos, 13566-585, Brazil
E-mail: {joueyama,jporto}@icmc.usp.br

Marcos Lordello Chaim
EACH, University of São Paulo (USP)
São Paulo, 03828-000, Brazil
E-mail: chaim@usp.br

1 Introduction

At present, a number of software development technologies (e.g. component-based approach, aspect-oriented programming, web services) can be employed for building systems that can be run on a variety of hardware platforms ranging from standard PCs to networked embedded devices. This scenario is also valid for reliable systems which are often required to run on a variety of hardware platforms including embedded devices. In this paper we are concerned with examining two types of heterogeneity:

- **Device heterogeneity.** Fault-tolerant systems are often deployed with heterogeneous devices which can range from PCs to embedded devices. However, this heterogeneity is expected to be adversely affected by the emergence of new hardware platforms.
- **Software language/middleware heterogeneity.** There are currently a large number of fault-tolerant policies each of which requires a particular procedure and strategy. They are normally based on heterogeneous programming languages and technology (e.g. publish-subscribe systems, web service applications, tuple spaces, message-oriented toolkits).

The aim of this paper is to investigate approaches that can lead to the development of middleware solutions that require different programming models in different environments. For this purpose, we introduce FlexFT, a generic framework for constructing reliable systems that can deal with both hardware and software heterogeneity. This consists of a minimal policy-free microkernel where fault tolerance policies are incremented as demanded. Furthermore, it provides a standardized and interoperable interface for sensor observations, by

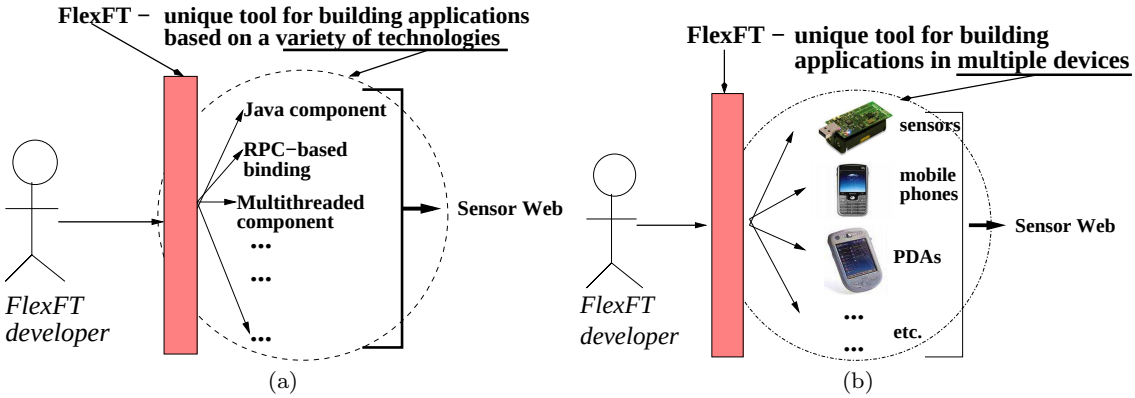


Fig. 1 FlexFT: Research Challenges

relying on the "Sensor Web" paradigm [4] established by the Open Geospatial Consortium (OGC).

The policy is deployed in the form of component plugins, which are destroyed when no longer required. FlexFT also supports dynamic adaptation, as these plugins can be reconfigured at runtime. The key feature of FlexFT is the provision of a unique tool for constructing reliable systems based on a wide range of software technologies targeted at a variety of hardware platforms. We evaluate FlexFT by means of case-studies and performance measurements; these show the following important benefits:

- **Flexibility.** Fault-tolerant systems can be developed and deployed independently of target platforms. The kernel can plugin the targeted platforms of a particular abstraction or behavior that is implemented.
- **Reusability/modularity.** The developers can reuse the existing components and processes that are employed for particular platforms.
- **Transference of skills.** The employment of different technologies to build applications for each targeted device and applicability does not allow the transfer of skills across different tools. Skill sets and areas of expertise are rarely transferable when they rely on different technologies. A generic approach can bring about the transference of skills because the developers only utilize a single tool for developing applications based on a variety of technologies.
- **Technology independency.** FlexFT allows heterogeneous components to be reconfigured e.g. both COM and Java components.
- **Interoperability.** FlexFT provides an interface that follows the "Sensor Web" paradigm, and makes it easy for developers to export sensor observations in accordance with the OGC standard *Sensor Observation Service* (SOS)[1]. In this manner, FlexFT provides interoperability with a variety of end-user applications that are able to consume data via web services that comply with the SOS standard.

The paper is structured in the following way: we set out by discussing the basic concepts and research challenges in Section 2. Section 3 reviews selected works related to the topics discussed in this paper. Following this, Section 4 outlines the generic framework for fault-tolerant system development; this includes an in-depth examination of the benefits obtained. Together with this section, Sections 5 and 6 examine some case studies which draw on the constructed prototype to show that the proposed approach attains an acceptable standard of performance and adequate resource consumption overhead. Finally, Section 7 concludes the paper and adds some comments about ongoing work.

2 Background and Research Challenges

Before examining the FlexFT architecture in detail, we will first discuss the determining factors underlying this new approach to reliable software development.

Sophisticated applications must now take account of a wide range of software technologies and middleware platforms to meet a large number of requirements. As illustrated in Fig. 1(a), a reliable system may have to use an implementation that has already been developed as Java or XPCOM[2] components and a multi-threaded component technology may also be necessary. In addition, multiple distribution abstractions may be required e.g. a publish-subscribe binding when the application is operating over ad-hoc wireless networks, along with Web Service middleware when the application needs to

[1] http://www.ogcnetwork.net/swe
[2] https://developer.mozilla.org/en/XPCOM

interact with a legacy service in the established infrastructure.

Furthermore, end users currently rely on a variety of devices ranging from PCs to smartphones. They are also often interested in accessing data from a number of sources including sensor motes (e.g. data coming from urban rivers, such as temperature and depth levels). Since these different sources have heterogeneous sensor protocols and interfaces, this scenario requires a good deal of effort on the part of application developers. Moreover, information systems that rely on sensors (e.g. information systems for disaster management) usually depend on integrating and composing several services, where each service handles different sensors that monitor and collect specific contextual information. Since typically the services are developed in an independent way, it is important do adhere to a standardized interface to ensure interoperability.

Against this backdrop, FlexFT offers a generic approach to help application developers mitigate the effects of this heterogeneity so as to ensure that end users can read data from sensors using a wide range of devices (e.g. PCs, smartphones and tablets). Rather than treat these as individual technologies that must be understood and integrated in an ad-hoc manner, FlexFT provides a single unified tool to allow developers to build applications to deal with the current heterogeneous environments which rely on the Sensor Web standards referred to above.

Reliable systems are also built on a diverse range of hardware platforms, as illustrated in 1(b). Instead of depending on porting applications across these platforms with the aid of the software technologies that are available, FlexFT offers a tool where the application can be developed independently of the target; the system then ensures that the appropriate implementation is deployed. Finally, since change is an essential feature of current systems, a uniform approach is required to provide dynamic software reconfiguration as well as to tackle the problem of heterogeneity.

The FlexFT framework provides mechanisms that make it easier for application designers to undertake the task of constructing reliable component-based systems. The construction of reliable systems is not a simple task; it requires the use of appropriate techniques during the whole software development cycle. In general, these techniques are based on the provision of redundancy (i.e. they make use of design diversity), both for error detection and error recovery.

Design diversity [2] means that multiple functionally equivalent software components are independently generated from the same initial specification. Two or more versions of the software component are independently developed from this specification, each by a group that does not interact with any other, and, whenever possible, employs different algorithms.

However, the provision of software redundancy involves the following: (i) an increase in the cost of creating the software, and (ii) a greater degree of complexity in the system, caused by the addition of redundant components. Ideally, the added software redundancy should be incorporated into the original system in a structured and non-intrusive manner to enable the application designers to construct dependable systems.

2.1 Recovery Block

Recovery Block [23] is a technique devised by Randell [22] from what, to some extent, was observed to be the practice at that time. The description outlined here has been slightly changed from the original description so that it is in accordance with the approach for component-based systems development. In a system with Recovery Blocks [23], the design of the system is broken down into fault recoverable blocks/modules (i.e. reliable system components). Each critical system component requires the separate development of alternative variants (modules of differing design aimed at a common specification) and one adjudicator to check the results produced by the variants (by means of an acceptance test). On entry to a recovery block, the state of the reliable system component (or of the whole system) must be saved to permit backward error recovery, i.e. to establish a checkpoint.

The primary *alternate* is executed and then the acceptance test is evaluated to provide an assessment of its outcome. If the acceptance test is passed, the outcome is regarded as successful and the recovery block can be exited. The information on the state of the component system obtained on entry (i.e. at the checkpoint) can be discarded. However, if the test fails or if any errors are detected by other means during the execution of the alternate, an exception is raised and backward error recovery is invoked. This restores the state of the component system to what it had been on entry. After this recovery, the next alternate is executed and then the acceptance test is applied again. This sequence continues until either an acceptance test is passed or all of the alternates have failed it. If all the alternates either fail the test or result in an exception (due to an internal error being detected), a failure exception will be signaled to the environment of the recovery block.

2.2 N-Version programming technique

Among the design diversity techniques, it is worth highlighting the N-Version programming technique [2]. In an N-Version software system, each module is formed of up to N different implementations. Each variant carries out the same task, but it is hoped in a different way. Each version then submits its answer to a *voter* or *decider* which determines the correct answer (for example, the majority of the votes) and returns this as the result of the N-Version component system.

There are few differences between the Recovery Block and the N-Version techniques, but they are important. In traditional Recovery Blocks, each alternative would be executed serially until an acceptable solution is found as determined by the adjudicator. The N-Version technique has always been designed to be executed in parallel. In a serial N-Version system, the cost in time of trying out multiple alternatives may be too expensive, especially for a real-time system. Another important difference in the two methods is the distinction between the roles of an adjudicator and decider. The Recovery Block technique requires each fault recoverable block (reliable system component) to build a specific adjudicator; in the N-Version technique, a single default decider (for example, the majority) may be used. On the basis of the assumption that the programmer can create a sufficiently simple adjudicator, the Recovery Block technique will create a system which is very unlikely to enter into an incorrect state. The engineering tradeoffs, especially monetary costs, involved with developing either type of system have both benefits and drawbacks, and it is important for the engineer to explore the space so as to be in a position to decide on what the best solution for his project should be.

2.3 State-based Variant Execution

The ability of dynamic reconfiguration — for example, to replace faulty components and/or to change the computation performed in fault situations — is a crucial factor in the development of reliable systems. When account is taken of the diversity of designs (components and their different variants), ideally the selection of the variant that will be executed should depend on the system and/or state of the component.

FlexFT supports dynamic adaptation, as these components and/or variants can be reconfigured at runtime, by implementing a slight variant of the State design pattern [13], a well known pattern that has been used in various applications. Its purpose is to allow an object to change its behavior when its internal state changes.

Consider the following example of motivation: components Sensor which represent individual sensors in a sensor network system and component Connection which provide the communication infra-structure for this system. For the sake of simplicity, we restrict Connection to two variants: Bluetooth and Wi-Fi. Depending on the current state of the system, it can respond in a different way to the client's (sensors) requests. For example, the implementation of a sendMessage operation depends on the state of the system: if the system is in its 'normal' state, the communication should be carried out by using Bluetooth (cheaper), otherwise, the Wi-Fi should be used. Moreover, the Connection component can change its current state when an event occurs or when a condition is satisfied (for example, after error detection or fault treatment procedures).

2.4 Sensor Web Enablement

As well as the issues referred to above, one of the main challenges for the application developer is how to integrate data that has been acquired from different types of sensors. Existing sensors use a large variety of sensor protocols (e.g. Sun SPOT ZigBee protocol, XBee/ZigBee, GumStix Wi-Fi) and sensor interfaces (e.g. nesC), and most applications are still dealing with this by integrating sensor resources through their own mechanisms. However, this manual bridging of the gap between sensor resources and applications leads to an extensive adaptation effort, and is considered to be a key cost factor in large-scale deployment scenarios [5].

This challenge to address the diversity of protocols, interfaces and sensor devices was addressed by the Open Geospatial Consortium (OGC) which in 2003 began to lay down a set of standards[3] with the aim of establishing the "Sensor Web" [4]. This can be defined as an infrastructure that allows the interoperable usage of sensor resources by ensuring that their discovery, access, tasking, as well as eventing and alerting, are carried out in a standardized way. Thus, the Sensor Web conceals the underlying layers, the network communication details, and heterogeneous sensor hardware, from the applications built on top of it, and thus allows users to share sensor resources more easily [5]. In the Sensor Web paradigm, all the sensors report their position and are available in the worldwide web; in addition, their metadata is registered so that they can all be uniformly accessed (and some of them even controlled) via the Internet [4].

The realization of the vision of sensor webs and networks is being pursued by the Sensor Web Enablement

[3] `http://www.ogcnetwork.net/swe`

(SWE) working group of OGC through the establishment of several (XML based) encodings for describing sensor resources and sensor observations, and through several standard interface definitions of web services. The first generation of SWE includes standards for [4]: a) description of sensor data; b) description of sensor metadata including properties and the behavior of the sensors; c) access to observations and sensor metadata based on standardized data formats and appropriate query and filter mechanisms; and d) setting of tasks for sensors to obtain measurement data.

FlexFT adopts OGC SWE standards to provide standardized access to sensor observations. The most important standard in this context is the Sensor Observation Service [18], which consists of a pull-based service for querying as well as inserting measured sensor data and metadata. FlexFT relies upon the existing open source SOS implementation from the 52° North Sensor Web framework[4] to provide the application developer with a service-oriented, standardized interface for sensor observations.

3 Related Work

This section presents the related work on component-based building system technology. We first review each platform and highlight their main features and contributions. Then, we outline how our work contributes towards the state of the art.

SaveCCM [15,1] is a component model designed to develop vehicular real-time systems. Within this domain, SaveCCM addresses the safety-critical subsystems responsible for controlling vehicular dynamics which includes power-train, steering and braking. However, SaveCCM only supports RTXC OS [21] and Microsoft Windows OSs, and thus is only deployable in environments where they are supported. Reconfiguration at runtime is not achieved in SaveCCM and hence, all the configurations are carried out at compile-time. This prevents the use of SaveCCM in systems that need a dynamic configuration such as a scenario in which new functionalities have to be deployed at runtime.

RUNES (Reconfigurable, Ubiquous, Networked Embedded Systems) [9,10] is a software platform aimed at providing the *software fabric* for developing networked embedded systems. It is based on a *component model* which encapsulates the characteristics of the devices and also allows the dynamic reconfiguration of the network of embedded systems. The component model is carried out by implementing a runtime API and the components themselves for particular devices. To support reconfiguration, the RUNES architecture employs meta-models which are updated by the API runtime whenever a component is created or destroyed. Although RUNES is able to handle changes occurring in the network of devices, fault tolerance techniques can only be supported at device level whereas the FlexFT middleware allows it to be handled at architecture level. In addition, FlexFT has an interface which makes sensor data available as a service in the Web in compliance with the OSG Sensor Web Enablement (SWE) standard [4]

The Loosely-coupled Component Infrastructure (LooCI) [16] is designed to support embedded Java ME (micro edition) platforms such as Sun SPOT or Java ME smart-phones. LooCI comprises an easy-to-use component model and a simple yet extensible networking framework. Each LooCI node is connected via a common event-bus communication substrate. Like other embedded component platforms, such as RUNES [10] or OpenCOM [12], LooCI components support runtime reconfiguration, concrete interface definitions, introspection and support for the re-wiring of bindings. LooCI was recently ported to a number of sensor devices and Android platforms, and is thus capable of creating component-based platforms in a heterogeneous environment. Unlike FlexFT, LooCI does not include a framework to create component-based runtime reconfigurable fault tolerant systems. Hence, it cannot create fault tolerant systems in a natural and explicitly-supported way.

The component-based operating system (OS) Lorien [19] allows users to experiment freely with software at any system level (e.g. MAC, drivers, routing, scheduling, etc.), and code can be (un)loaded dynamically during experiment runtime without resetting the nodes. The OS can also be used as a boot manager to run other Wireless Sensor Network (WSN) OSs of the users's choice. Lorien runs on T-Mote class devices and provides all the benefits of OpenCOM while running on resource constrained devices. It is particularly targeted at providing runtime reconfiguration (flexibility) for OSs that runs on WSNs. While it provides flexibility on WSN OSs, Lorien does not provide a generic framework for constructing reconfigurable fault-tolerant systems. Furthermore, Lorien does not provide an implementation that can ensure there will be the interaction with the sensor web paradigm.

The middleware developed in the context of the MORE project (Network-centric *M*iddleware for Gr*O*up communication and *R*esource Sharing across Heterogeneous *E*mbedded Systems) [27] targets heterogeneous embedded systems in the Service Oriented

[4] `http://52north.org/swe`

Architecture (SOA) context. MORE middleware allows XML-based information (e.g., SOA data, XML-base policies) to be transferred to embedded services nodes in an efficient manner. The idea is to reduce consumption of resources (e.g., battery, processing time) in the devices. To achieve such a goal, the μSOA approach is proposed to reduce the message size and parsing overhead. According to the authors, a μSOA message requires 2.5% of a Standard SOAP message. In contrast, FlexFT is a middleware system supporting the development of fault-tolerant components with a service-oriented interface. It does not address the question of inter-node communication.

In summary, we argue that FlexFT provides a framework for constructing runtime reconfigurable component-based fault tolerant systems. Given that the FlexFT kernel itself is minimal, it is deployable in a variety of devices including the sensor motes. We also argue that none of the platforms discussed above adopt SWE standards to ensure that data from the sensor nodes are accessible via the web by any end user interested in this kind of information.

4 FlexFT Framework

The FlexFT framework architecture can be decomposed into two layers and a rough draft of its structure is illustrated in Fig. 2(a):

- **Fault-Tolerant Component Frameworks.** This layer is responsible for providing mechanisms for developing reliable component-based systems. These mechanisms are implemented in the form of component frameworks. Component Framework (CF) has been defined as "*collections of rules and interfaces that govern the interaction of a set of components 'plugged into' them*" [25]. A CF embodies rules and interfaces that make sense in a specific application domain.
- **Component Runtime Kernel Layer.** This layer provides support for the development of component-based reliable systems. That is, this layer provides the inherent component model operations of FlexFT such as: (i) Loading components and instantiation, (ii) Definition of receptacles and (iii) Definition of bindings.

4.1 FlexFT: Component Model

The FlexFT framework provides mechanisms (implemented as Component Frameworks) that make easier the task of constructing reliable component-based systems by application designers. Fig. 2(b) shows the FlexFT component approach that is employed to implement the reliable (redundant) components and their respective variants.

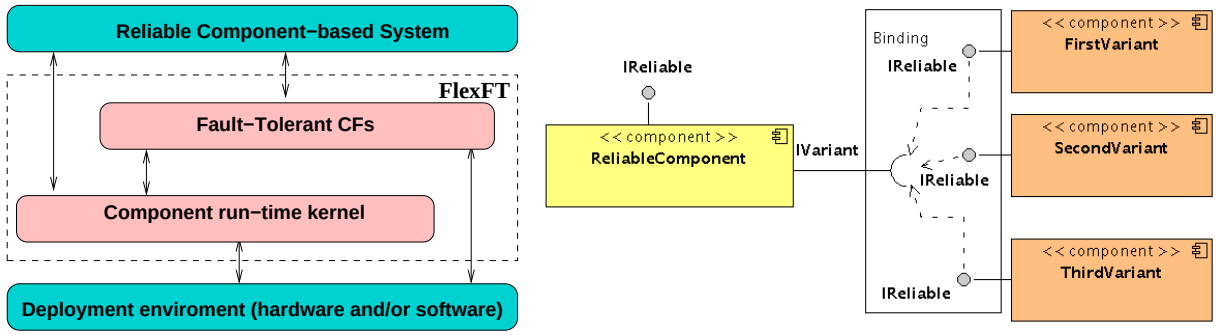
- The FirstVariant, SecondVariant and ThirdVariant components consist of variants (multiple functionally equivalent software components that are independently generated from the same initial specifications).
- The component ReliableComponent is a controller that is responsible for coordinating the execution of the variants and invoking the inherent operations (acceptance test, adjudication and so on) of different design diversity techniques.
- The Binding mechanism connects both the provided and required interfaces. It is worth pointing out that the granularity of this connection is N provided interfaces (IReliable) to 1 required interface (IVariant).

4.2 FlexFT framework classes

Fig. 3 shows the classes and interfaces that comprise the FlexFT implementation of three different design diversity techniques. Fault-tolerant systems designers must extend these classes and/or implement these interfaces to build reliable systems. When the FlexFT prototype was implemented, it was based on the OpenCOMJ — the OpenCOM [8][12] implementation in Java. OpenCOM is a lightweight, efficient and reflective component model.

The abstract class OpenCOMComponent includes generic source code for OpenCOM components (OpenCOMJ API). The abstract class FTComponent extends the OpenCOMComponent class and is the root of the reliable (redundant) components hierarchy and implements the basic life-cycle operations (creation, destruction, connection, disconnection) of reliable OpenCOM components. Moreover, this class has a reference to the interface Receptacle (OpenCOMJ implementation of required interfaces). FTComponent subclasses use different implementations of this interface and take into account the inherent characteristics of each fault-tolerance technique (N-Version Programming, Recovery Blocks and State-based execution).

The FTInterface provides the operation execute (String methodName, Object[] params). This operation is implemented by FTComponent subclasses: RBComponent (Recovery Blocks technique), NVComponent (N-Version Programming technique) and ContextComponent (State-based Variant). The execute operation is responsible for executing the variants (using the reference to the interface Receptacle) and takes into account



(a) FlexFT Architecture (b) FlexFT Reliable Component

Fig. 2 FlexFT Framework

the inherent characteristics of each technique (sequential execution, parallel execution and so on).

5 Case Study One: Design diversity techniques

This section provides some simple examples that illustrate how the FlexFT framework can be used to implement reliable components using different design diversity techniques [2].

5.1 N-Version Programming Technique

We examine the implementation of a simple reliable component based on the N-Version Programming technique [2] using the FlexFT framework.

Fig. 4 shows the MultiplyNV component (and its variants AddVariant, MultiplyVariant and WrongVariant) that provide the functionality of returning the product of two integers. The MultiplyNV class represents the ReliableComponent (Fig. 2) and extends the NVComponent abstract class (Fig. 3) that implements the execute method by taking into account the inherent features of the N-Version programming technique. That is, this method is responsible for executing the variants and for obtaining the respective decision (NVInterface decide method). It is worth mentioning that the FlexFT framework provides a default algorithm for the result decision (majority) through the NVComponent decide method implementation. However, NVComponent sub-classes can reimplement this method and hence use a different result decision algorithm.

In addition, the MultiplyNV class implements the IMultiply interface (the functionality of returning the product of two integers). This implementation (Code 1) simply consists of invoking the execute method discussed earlier.

The AddVariant, MultiplyVariant and WrongVariant represent three implementations of the functionality of returning the product of two integer numbers (IMultiply interface). The AddVariant class only uses addition operations (operator +) to implement this functionality, while the MultiplyVariant class uses the operation of multiplication (operator *). The WrongVariant class always returns the value 0 (no matter what the parameter values are). Thus, the result of this erroneous variant will always be disregarded because the other variants always return correct values.

Code 1 MultiplyNV: multiply method

```
public Integer multiply(Integer a, Integer  b) {
  Object[] params = new Object[]{a, b};
  return (Integer) this.execute("multiply", params);
}
```

5.2 Recovery Block Technique

This section gives an example that illustrates how the FlexFT framework can be used to implement reliable components based on the Recovery Block technique [23].

Fig. 5 shows the SortRB component (and its NothingSort, InvertionSort and QuickSort variants) which provide the functionality of sorting an array of integers in ascending order.

The SortRB class represents the ReliableComponent (Fig. 2). This class extends the RBComponent class (Fig. 3) and implements the inherent Recovery Block operations: acceptance test, component state saving and restoration. In this example, the acceptance test has to check the following condition: V is an array of integers, $V[i+1] \geq V[i]$ for i = 1,2, ... , n-1. It is worth mentioning that the FlexFT framework provides a default algorithm (based on object serialization) for the component state saving and restoration through the RBComponent saveState and restoreState methods implementation. However, RBComponent sub-classes can

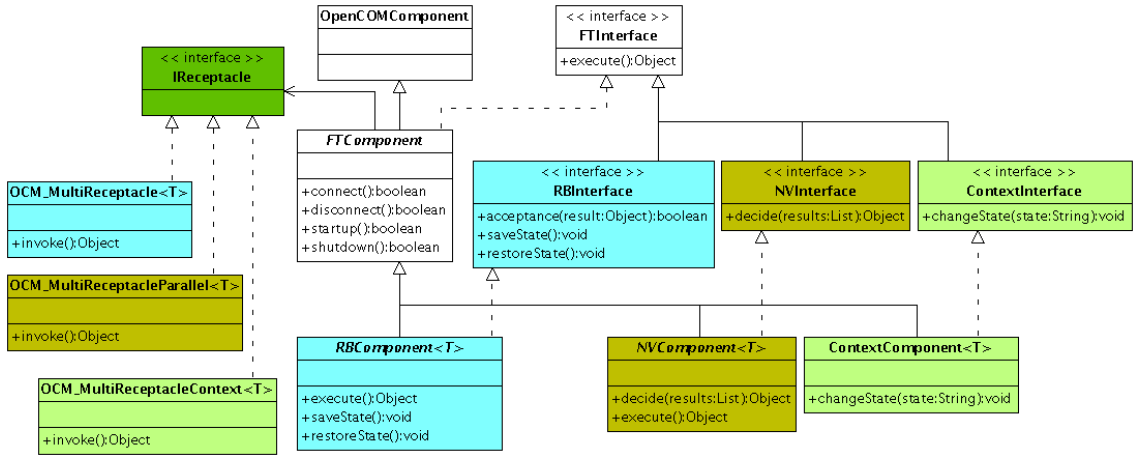


Fig. 3 FlexFT Framework classes

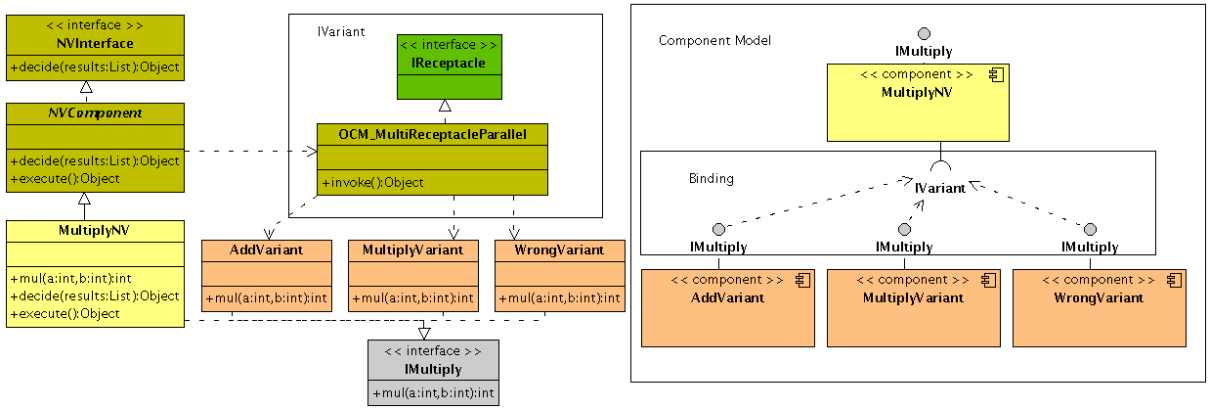


Fig. 4 N-Version Programming realization

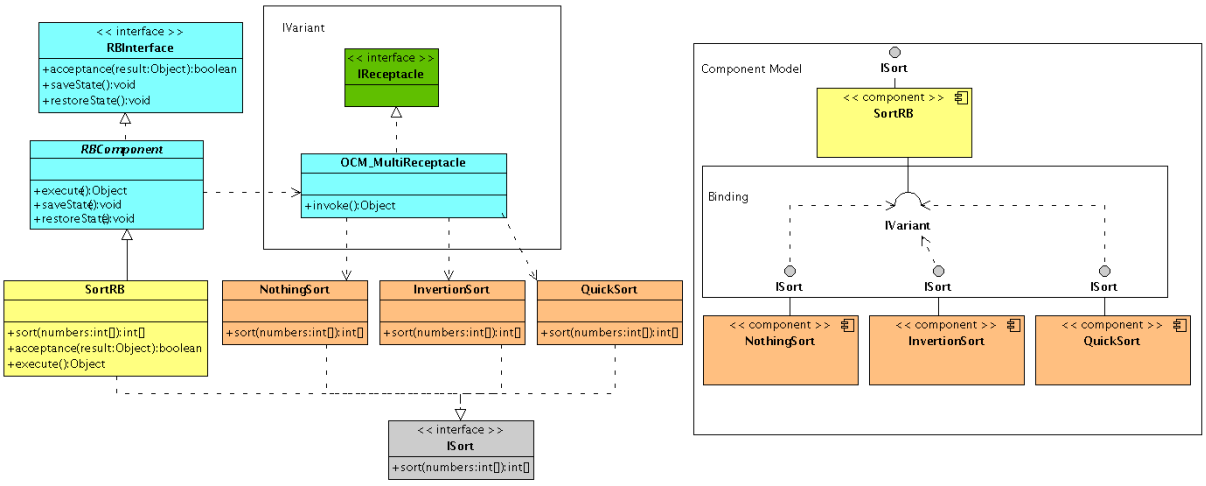


Fig. 5 Recovery Block realization

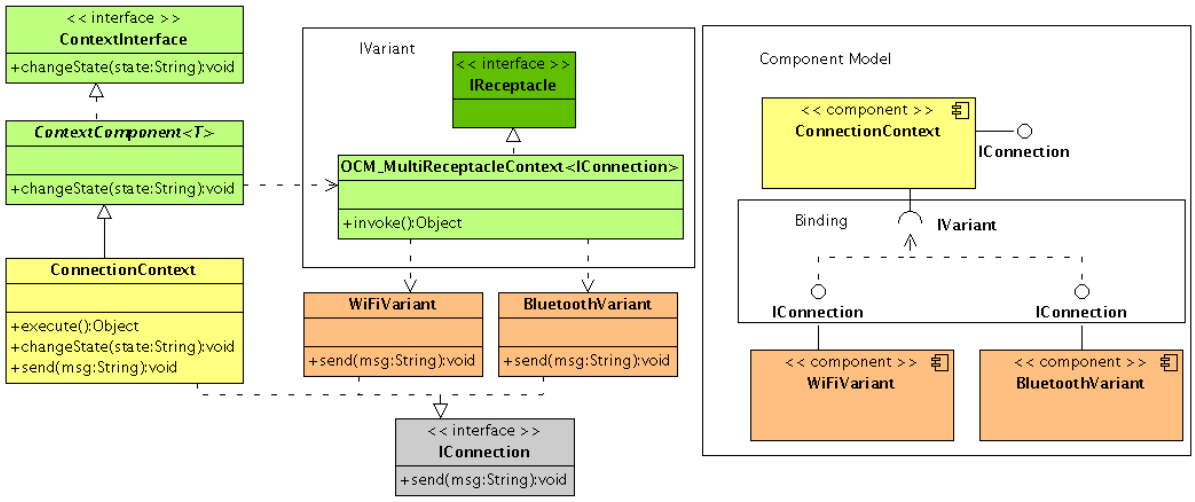


Fig. 6 State-based Variant Execution realization

reimplement these methods and hence use a different component state saving and restoration technique.

In addition, the SortRB class implements the ISort interface (the functionality of sorting an array of integers in ascending order). This implementation (Code 2) consists simply of invoking the execute (from FTInterface) method discussed earlier.

Code 2 SortRB: sort method

```
public int[] sort(int[] elements) {
  return (int[]) this.execute("sort", elements);
}
```

The NothingSort, InvertionSort and QuickSort represent three implementations of the functionality of sorting an array of integers (interface ISort) in ascending order. It should be said that these classes should extend the OpenCOMComponent class discussed above.

- The NothingSort class simply returns the array passed as a method parameter. If this array is already sorted, this variant will pass the acceptance test.
- The InvertionSort class reverses the array passed as a method parameter. If the array is sorted in descending order, this variant will pass the acceptance test.
- The QuickSort class sorts the array by means of the QuickSort algorithm. It is expected that this variant will always pass the test of acceptance.

5.3 State-based Variant Execution

This section gives a simple example to illustrate how the FlexFT framework can be used to implement reliable components by using the State-based variant execution. Fig. 6 shows the ContextConnection component (and its variants WiFiVariant and BluetoothVariant) that provide the functionality of sending messages by means of different wireless network infra-structures (Wi-Fi or Bluetooth).

The ContextConnection class represents the ReliableComponent (Fig. 2). This class extends the ContextComponent class (Fig. 3) and implements the interface IConnection. This implementation (Code 3) simply consists of invoking the execute (from FTInterface) method discussed earlier.

Code 3 ContextConnection: state-based send method

```
public void send(String  message) {
  Object[] args = new Object[] { message };
  this.execute("send", args);
}
```

The WiFiVariant and BluetoothVariant represent two implementations of the functionality of sending a message (interface IConnection):

- The WiFiVariant class sends messages using the Wi-Fi technology and
- The BluetoothVariant class sends messages using the Bluetooth technology.

5.4 Experimental results

The N-Version programming technique [2] (example discussed earlier) was implemented (together with other

design diversity techniques) and deployed in two different hardware platforms: Standard PC and Sun SPOT (Sun Small Programmable Object Technology)[5].

The experiment was run in a desktop with an Intel i5 CPU 2.67 GHz processor and 8 GBytes of RAM memory running Ubuntu 12 operational system. Sun SPOT is a wireless sensor network (WSN) mote developed by Sun Microsystems. Unlike other available mote systems, Sun SPOT is built on the Squawk Java Virtual Machine [7]. For comparative purposes, the Squawk Java Virtual Machine was used in both platforms. The application code used to assess the cost of utilizing the FlexFT framework is listed in Appendix A.

Performance and Resource Consumption	PC	Mote*
load/instantiate NVComponent (ms)	6.2	110.1
load/instantiate NVComponent (bytes)	1472	1472
load/instantiate Variants (ms)	7.2	196.6
load/instantiate Variants (bytes)	3004	3004
redundant operation execution (ms)	1.3	10.8
runtime dynamic reconfiguration (ms)	1.2	30.1

*Sun SPOT (Sun Small Programmable Object Technology)

Table 1. FlexFT Evaluation

Table 1 shows the average performance (measured in ms) and the memory consumption (measured in bytes) of the main operations of the N-Version programming technique: (a) to load and instantiate NVComponents, (b) to load and instantiate the variants, (c) to execute a redundant operation — i.e. executing the Variants and the method for the respective (majority) decision result and (d) to unload a component and after that, load another similar component — i.e. dynamic runtime reconfiguration.

On the basis of these values, it can be argued that the proposed approach has an acceptable performance and resource consumption overhead across heterogeneous platforms. It should be stressed that these values are in compliance with those of the study conducted by Salmony [24] which states that the reconfiguration delays should not exceed 250ms. Moreover, according to [17] for multimedia applications, delays less than 150 ms are not even noticeable and the maximum tolerable delay is 400 ms.

6 Case Study Two: Sensor Web Enablement

This section gives an example that illustrates how the FlexFT framework can be used to provide standardized access to sensor observations while automatically disregarding potentially erroneous readings. First, the example scenario is described, and in the sequence the experimental results achieved are presented.

6.1 Example

The example setting employed is illustrated in Fig. 7 and described as follows.

- A set (3 to 5) of Sun SPOT sensors which work together to monitor the temperature from the surrounding environment. Each sensor uses a different port number (in the range of 66 to 70) to broadcast, once every second, the collected temperature data. Owing to its simplicity, the adopted communication protocol is the radiogram protocol that provides datagram-based communication (with no guarantees about delivery or ordering) between two devices.
- By default, the 52° North Sensor Web Framework does not provide the observations of the temperature sensors. That is, it was necessary to register the temperature sensor as follows: (a) create a SensorML[6] instance document that describes this kind of sensor; (b) store the instance document in the configuration directory of 52° North Sensor Web Framework.
- The basestation employs a N-Version programming approach to collect the information that has been broadcast. In other words, the basestation instantiates the SensorNV (and its variants) that provide the functionality of returning the consensual temperature from the sensors. Each SensorVariant is bound to a specific Sun SPOT sensor since it listens to a specific port number (in the range 66 to 70). Table 2 shows the relationship between the Sun SPOT sensors and the variants. Each Sun SPOT is listed by its IEEE network number.

Sun SPOT Sensor	Port	Variant
0014.4F01.0000.132D	66	SensorVariantOne
0014.4F01.0000.2BCC	67	SensorVariantTwo
0014.4F01.0000.2A56	68	SensorVariantThree
0014.4F01.0000.2470	69	SensorVariantFour
0014.4F01.0000.4432	70	SensorVariantFive

Table 2. Sun SPOT Sensors & Variants

- In order to validate the N-Version programming technique, some erroneous temperature values were injected (by placing a heat source close to the sensors). As expected, these results were disregarded because the correct temperature values were returned by other sensor/variants.

[5] http://www.sunspotworld.com/docs/Red/spot-developers-guide.pdf

[6] http://www.opengeospatial.org/standards/sensorml

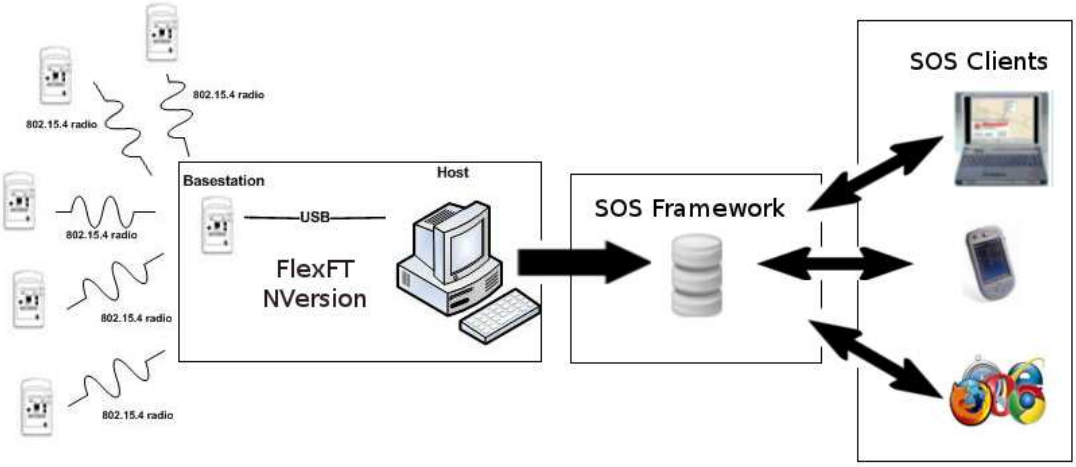


Fig. 7 FlexFT Sensor Web Enablement

– The next step is to store the consensual temperature (and corresponding metadata useful for discovery and human assistance) in the 52° North Sensor Web Framework database.
– As discussed earlier, the 52° North Sensor Web Framework provides Sensor Web Services which can be accessed by different types of clients: desktop, mobile or web applications.

6.2 Experimental results

To perform our experiments with FlexFT and the Sensor Web Enablement (SWE), we utilized the same hardware configuration described in Section 5.4 and the 52° North Sensor Web Framework assessing a PostgreSQL[7] data base using the JDBC api[8]. The application code utilized to illustrate FlexFT being utilized in the SWE context is listed in Appendix A.

Unlike the previous implementation, the N-Version programming technique [2] was only implemented and deployed in the Sun SPOT basestation. Three scenarios were employed to evaluate this implementation. The one single difference between these three scenarios is the number (three to five) of sensors/variants employed. The first experiment employs three sensors, while the second and third experiments employ four and five sensors respectively.

Table 3 shows the average performance of these three experiments; that is, the average of the execution for 1000 samples where each sample represents the performance (measured in ms) of the main operations of the N-Version programming technique combined with the Sensor Web enablement approach. The following operations were assessed:

– To execute a redundant operation — i.e. to execute the SensorVariants (collect the temperature of different sensors),
– To execute the method for the respective result of the (majority) decision and
– To store the consensual temperature (and corresponding metadata that is useful for discovery and human assistance) in the 52° North Sensor Web Framework database.

Experiment	No. of Sensors	Average Performance (ms)
1^{st}	3	43.426
2^{nd}	4	46.856
3^{rd}	5	52.188

Table 3. Performance of the Experiments

On the basis of these values, the difference in performance between the three experiments suggests the approach performance increases linearly with the growth of sensor nodes. However, the analysis of the data shows that there is a great variation in the sample performance. That is, several outliers were observed while the results were being obtained. This might be due to the overhead inherent to the Java Virtual Machine (JVM) such as Garbage Collection.

7 Concluding remarks

This paper discussed the use of a generic component-based framework for the construction of adaptive fault tolerant systems that can integrate and re-use technologies and deploy them across heterogeneous devices. We

[7] http://www.postgresql.org/
[8] http://www.oracle.com/technetwork/java/javase/jdbc/index.html

have implemented a framework prototype and evaluated the potential benefits by means of two case studies and performance measurements. These show that the proposed framework can deal with a wide degree of heterogeneity with minimal resource overheads.

With regard to our generalized approach, it should be emphasized that FlexFT was designed to construct fault-tolerant systems for a variety of platforms including PCs and sensor motes. Though our prototype was only employed to build fault-tolerant systems in devices such as sensors and PCs, we believe that sensors are the most difficult devices to be programmed. This is also recognized by the research community (i.e. devices that are very hard to be programmed [20]). Thus, since FlexFT was adopted to program this resource-constrained device, we believe that FlexFT can be easily adopted to construct fault-tolerant systems in other devices such as smartphones and PDAs.

The generality of hardware and software are achieved by means of the so-called loader and binder extension plugins [12] that we borrowed from OpenCOM. In short, the loader plugin encapsulates the complexity of loading software in a particular deployment environment (e.g. loader for a Assembly based software component into the Sun SPOT sensor mote or a loader for deploying N-version system based on Java Multithreads). The binder plugin provides a wide range of 'binding mechanisms'. Using binders, developers are free to implement a wide range of binding mechanisms that might be required in the underlying deployment environment. For example, he/she may implement a binder that creates connections between Java components or a binder that connects components written in Assembly language. That way, one can create fault tolerant software for a variety of environment such as sensor nodes, mobile phones and desktop PCs.

With regard to future studies, two different directions can be envisaged:

Fault tolerance techniques. Regarding the examples discussed in Section 5, we plan to incorporate other fault tolerance techniques into the FlexFT framework such as coordinated atomic action [6][28], concurrent exception handling [14], context-based exception handling [3][26] and so on. Moreover, we plan to evaluate how the FlexFT framework can be fitted into the context of critical embedded systems development.

Multi-hop communication. Regarding the example discussed in Section 6, we plan to utilize the FlexFT framework in the implementation of multi-hop communication in this scenario. The multi-hop communication [11] is the best choice of economy power consumption in wireless sensor networks (WSNs), since the energy required for communication between two arbitrary nodes **A** and **B** depends on the distance between the two nodes. In this scenario, the FlexFT framework will be employed to implement the sensors and the base station.

Acknowledgments

The authors would like to express their gratitude for the support granted by CNPq and FAPESP to the INCT-SEC (National Institute of Science and Technology — Critical Embedded Systems - Brazil), processes 573963/2008-9 and 08/57870-9.

Dr. Delano Beder and Dr. Jó Ueyama are also grateful to CNPq for the support provided for the REACT project (process 483699/01881-5).

Dr. Jó Ueyama would also like to thank FAPESP (process 2008/05346-4), CNPq (process 474803/2009-0) and RNP (CIA2-RIO) for their financial support.

Dr. João P. de Albuquerque and Dr. Jó Ueyama are also grateful for the support granted by FAPESP (process 2008/58161-1).

Finally, Dr. João P. de Albuquerque would also like to thank the Alexander von Humboldt Foundation for its sponsorship.

References

1. Åkerholm, M., Möller, A., Hansson, H., Nolin, M.: SAVE-Comp - a Dependable Component Technology for Embedded Systems Software. Technical Report MDH-MRTC-165/2004-1-SE, Målardalen University (2004)
2. Avizienis, A.: The N-Version Approach to Fault-Tolerant Software. IEEE Trans. Software Eng. **11**(12), 1491–1501 (1985)
3. Beder, D., de Araújo, R.B.: Towards the Definition of a Context-Aware Exception Handling Mechanism. In: Dependable Computing Workshops (LADCW), Fifth Latin-American Symposium on Dependable Computing (LADC'2011), pp. 25–28 (2011)
4. Botts, M., Percivall, G., Reed, C., Davidson, J.: OGC sensor web enablement: Overview and high level architecture. In: F. Fiedrich, B.V. de Walle (eds.) Proceedings of the 5th International ISCRAM Conference, May 2008, pp. 713–723. Washington, DC, USA (2008)
5. Broering, A., Echterhoff, J., Jirka, S., Simonis, I., Everding, T., Stasch, C., Liang, S., Lemmens, R.: New generation sensor web enablement. Sensors **11**(3), 2652–2699 (2011)
6. Capozucca, A., Guelfi, N., Pelliccione, P., Romanovsky, A., Zorzo, A.: Frameworks for Designing and Implementing Dependable Systems using Coordinated Atomic Actions: A Comparative Study. Journal of Systems and Software **82**(2), 207–228 (2009)
7. Cifuentes, C.: Squawk - A Java VM for Small (and Larger) Devices (2005). Slides for the IFIP WG 2.4 Meeting

8. Clark, M., Blair, G., Coulson, G., Parlavantzas, N.: An Efficient Component Model for the Construction of Adaptive Middleware. In: IFIP Middleware. Germany (2001)
9. Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., Picco, G.P., Sivaharan, T., Weerasinghe, N., Zachariadis, S.: The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In: 5th IEEE International Conference on Pervasive Computing and Communications (PerCom 2007), pp. 69–78. IEEE Computer Society, Los Alamitos, CA, USA (2007)
10. Costa, P., Coulson, G., Mascolo, C., Mottola, L., Picco, G., Zachariadis, S.: A Reconfigurable Component-based Middleware for Networked Embedded Systems. International Journal of Wireless Information Networks **14**(2), 149–162 (2007)
11. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed systems - Concepts and Designs, 3rd edn. International Computer Science Series. Addison-Wesley-Longman (2002)
12. Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T.: A Generic Component Model for Building Systems Software. ACM Transaction on Computer Systems **26**(1) (2008)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley Publishing (1995)
14. Garcia, A., Beder, D., Rubira, C.: A Unified Meta-Level Software Architecture for Sequential and Concurrent Exception Handling. Computer Journal **44**, 569–587 (2001)
15. Hansson, H., Akerholm, M., Crnkovic, I., Torngren, M.: SaveCCM – A Component Model for Safety-critical Realtime Systems. In: Euromicro Conference, Special Session Component Models for Dependable Systems. IEEE (2004)
16. Hughes, D., et. al.: Looci: a loosely-coupled component infrastructure for networked embedded systems. In: MoMM'2009 - The 7th International Conference on Advances in Mobile Computing and Multimedia, pp. 195–203 (2009)
17. Kurose, J., Ross, K.: Computer Networking: A Top-Down Approach. Addison-Wesley Publishing (2009)
18. Na, A., Priest, M.: Ogc implementation specification 06-009r6: Opengis sensor observation service (sos) (2007)
19. Porter, B., Coulson, G.: Lorien: a pure dynamic component-based operating system for wireless sensor networks. In: MidSens '09: Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks, pp. 7–12. ACM, New York, NY, USA (2009)
20. Porter, B., Roedig, U., Coulson, G.: Type-safe updating for modular wsn software. In: Distributed Computing in Sensor Systems, 7th IEEE International Conference and Workshops, DCOSS 2011, pp. 1–8 (2011)
21. Quadros, S.: RTXC Kernel User's Guide. `http://www.quadros.com`
22. Randell, B.: System Structure for Software Fault Tolerance. IEEE Transactions on Software Engineering **1**(2), 220–232 (1975)
23. Randell, B., Xu, J.: Trends in Software, chap. The Evolution of the Recovery Block Concept, pp. 1–22. Wiley (1994)
24. Salmony, M., Stuttgen, H.: Transport Services for Multimedia Applications on Broadband Networks. Computer Communications **13**(4), 197–203 (1990)
25. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, second edn. Addison-Wesley (2002)
26. Tripathi, A., et al.: Exception Handling Issues in Context-Aware Collaboration Systems for Pervasive Computing. In: LNCS Advanced Topics in Exception Handling Techniques, vol. 4119, pp. 161–180. Spring (2006)
27. Wolff, A., Michaelis, S., Schmutzler, J., Wietfeld, C.: Network-centric middleware for service oriented architectures across heterogeneous embedded systems. In: IEEE 11th International EDOC Conference Workshop. EDOCW '07, pp. 105–108. IEEE Computer Society, Los Alamitos, CA, USA (2007)
28. Xu, J., Randell, B., Romanovsky, A., Rubira, C., Stroud, R., Wu, Z.: Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In: 25th International Symposium on Fault Tolerant Computing (FTCS'25), pp. 499–508 (1995)

Appendix

A – Application code utilized in the case studies

The application code implemented for the case studies discussed in Sections 5.4 and Section 6.2 is listed below.

```
// File: TemperatureDemoHostApplication.java

package org.sunspotworld.demo;

import FlexFT.NVInterface;
import OpenCOMV2.OCM_InterfaceRefList;
import OpenCOMV2.OpenCOMV2;
import org.sunspotworld.demo.sensor.
    Temperature;
import org.sunspotworld.demo.sensor.ISensor;
import org.sunspotworld.demo.sensor.SensorNV
    ;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.SQLException;
import java.util.Calendar;
import java.util.Date;
import com.sun.spot.util.Utils;
import java.io.PrintWriter;
import java.sql.Connection;

public class TemperatureDemoHostApplication
    {

    private static final String DATABASE_URL
        = "jdbc:postgresql://localhost
        :5432/";
    private static final String
        DATABASE_NAME = "SosDatabase";
    private static final String
        DATABASE_USER = "postgres";
    private static final String
        DATABASE_PASSWORD = "postgres";
    private static final String JDBC_DRIVER
        = "org.postgresql.Driver";
    // Name of the database table where we
        store sensor readings
```

```
    private static final String
        DATA_TABLE_NAME = "observation";
    private Statement stmt = null;
    private Connection dbCon = null;
    private ISensor pISensor;
    private final int NRO_SENSORS = 5;
    private final int NRO_SAMPLES = 1000;

    private void run() throws Exception {
        try {
            setUp();
            collectData();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            tearDown();
        }
    }

    private void setUp() throws Exception {

        final String className = "org.
            sunspotworld.demo.sensor.Sensor"
            ;


        System.out.println("Database demo
            application starting ...");
        try {
            // Register the JDBC driver for
                mysql/postgres
            // This is typically found in a
                JAR file
            // named something like
            // mysql-connector-java-<version
                >-bin.jar
            // which should be in the
                classpath
            // See user.classpath in build.
                properties
            // for this Sun SPOT host
                application
            Class.forName(JDBC_DRIVER);

            // Define URL of database server
                for
            String url = DATABASE_URL +
                DATABASE_NAME;

            // Get a connection to the
                database for given user/
                password
            dbCon = DriverManager.
                getConnection(url,
                DATABASE_USER,
                    DATABASE_PASSWORD);

            // Display URL and connection
                information
            System.out.println("URL: " + url
                );
            System.out.println("Connection:
                " + dbCon);

            // Get a Statement object
            stmt = dbCon.createStatement();

            NVInterface pINV;
            OCM_InterfaceRefList pintfs;

            OpenCOMV2 runtime = OpenCOMV2.
                getInstance();
            final String sensorInterface =
                ISensor.class.getName();
            final String nvInterface =
                NVInterface.class.getName();

            // Create NVSensor

            int sensorNVId = runtime.load(
                SensorNV.class.getName());
            int sensorNVComId = runtime.
                instantiate(sensorNVId);

            pintfs = (OCM_InterfaceRefList)
                runtime.getprop(
                sensorNVComId, "INTERFACES")
                ;
            pISensor = (ISensor) pintfs.
                getInterfaceRef(
                sensorInterface);
            pINV = (NVInterface) pintfs.
                getInterfaceRef(nvInterface)
                ;
            String[] sensores = new String[
                NRO_SENSORS];

            for (int i = 1; i <= NRO_SENSORS
                ; i++) {
                sensores[i - 1] = Class.
                    forName(className + i).
                    getName();
            }
            pINV.configure(sensorNVComId,
                sensores);

            System.out.println("setUp
                completed successfully");
        } catch (Exception e) {
            System.err.println("setUp caught
                " + e);
            System.err.println("Make sure
                that mySQL is installed
                properly \n"
                    + "and has a Test
                        database accessible
                        via the \n"
                    + "default administrator
                        settings on
                        localhost:3306\n");
            throw e;
        }
    }

    public double getTemperature() {

        Temperature result = pISensor.
            getTemperature();

        return result.getValue();
    }
```

```
public void collectData() throws
    Exception {
    String id = null;
    String ts = null;
    double val = 0;


    String[] saida = new String[
        NRO_SAMPLES];

    System.out.println("Sampling ...\n[
        Each * indicates one sample, ! 
        implies"
            + " radio timed out waiting 
                for a sample]");

    // Main data collection loop
    for (int i = 1; i <= NRO_SAMPLES; i
        ++) {
        try {

            Utils.sleep(1000);

            System.out.println("* " + i)
                ;

            long start = System.
                currentTimeMillis();

            val = getTemperature();

            Calendar cal = Calendar.
                getInstance();
            cal.setTime(new Date(System.
                currentTimeMillis()));
            ts = cal.get(Calendar.YEAR)
                + "-"
                    + (1 + cal.get(
                        Calendar.MONTH))
                        + "-"
                    + cal.get(Calendar.
                        DAY_OF_MONTH) +
                        " "
                    + cal.get(Calendar.
                        HOUR_OF_DAY) + "
                        :"
                    + cal.get(Calendar.
                        MINUTE) + ":"
                    + cal.get(Calendar.
                        SECOND);

            final String proc_id = "urn:
                ogc:object:feature:
                Sensor:IFGI:ifgi-sensor
                -3";
            final String foi_id = "
                foi_3001";
            final String phe_id = "urn:
                ogc:def:phenomenon:OGC
                :1.0.30:temperature";
            final String off_id = "
                TEMPERATURE";

            String query = "INSERT INTO 
                " + DATA_TABLE_NAME + "(
                    time_stamp, procedure_id
                    ,";
                query += "
                    feature_of_interest_id, 
                    phenomenon_id,
                    offering_id,
                    numeric_value)";
                query += " VALUES('" + ts +
                    "','" + proc_id + "','"
                    + foi_id + "',";
                query += "'" + phe_id + "','
                    " + off_id + "','" + val
                     + "')";

                stmt.executeUpdate(query);

                long end = System.
                    currentTimeMillis();

                saida[i - 1] = i + " " + (
                    end - start) + " " + val
                    ;

                System.out.println(saida[i -
                     1]);


            } catch (SQLException e) {
                System.out.println("Caught "
                     + e
                        + " while storing 
                            sensor sample <"
                        + "\'" + id + "\',\'
                            " + ts + "\'," +
                             val + ">");
                throw e;
            } catch (Exception e) {
                System.out.println("Caught "
                     + e
                        + " while reading 
                            sensor samples."
                            );
                throw e;
            }
        }
        PrintWriter writer = new PrintWriter
            (String.valueOf(NRO_SENSORS) + "
            .csv");
        for (String linha : saida) {
            writer.println(linha);
        }
        writer.close();
    }

    public void tearDown() throws Exception
        {
        if (dbCon != null) {
            dbCon.close();
        }
        System.exit(0);
    }

    public static void main(String[] args)
        throws Exception {
        TemperatureDemoHostApplication app =
             new
```

```
            TemperatureDemoHostApplication()
            ;
        app.run();
    }
}

//File: Temperature.java

package org.sunspotworld.demo.sensor;

public class Temperature {

    private double value;

    public Temperature(double value) {
        this.value = value;
    }

    public double getValue() {
        return value;
    }

    public String toString() {
        return Double.toString(value);
    }

    public boolean equals(Object obj) {
        boolean result = false;
        if (obj instanceof Temperature) {
            Temperature temp = (Temperature)
                obj;
            result = Math.abs(temp.value -
                this.value) <= 1;
        }
        return result;
    }

    public int hashCode() {
        return (int) value;
    }
}

// File: ISensor.java

package org.sunspotworld.demo.sensor;

public interface ISensor {
    public Temperature getTemperature();
}

// File: SensorNV.java

package org.sunspotworld.demo.sensor;

import FlexFT.NVComponent;
import OpenCOMV2.IConnections;

public class SensorNV extends NVComponent
    implements IConnections, ISensor {

    public String getRequiredInterface() {
        return ISensor.class.getName();
    }

    public Temperature getTemperature() {
        return (Temperature) this.execute(
            new Object[]{null});
    }
}

// File: Sensor1.java

package org.sunspotworld.demo.sensor;

public class Sensor1 extends Sensor {

    public int getPort() {
        return 67;
    }
}


// File: Sensor2.java

package org.sunspotworld.demo.sensor;

public class Sensor2 extends Sensor {

    public int getPort() {
        return 68;
    }


}

// File: Sensor3.java

package org.sunspotworld.demo.sensor;


public class Sensor3 extends Sensor {

    public int getPort() {
        return 69;
    }
}


// File: Sensor4.java

package org.sunspotworld.demo.sensor;

public class Sensor4 extends Sensor {

    public int getPort() {
        return 70;
    }
}

// File: Sensor5.java

package org.sunspotworld.demo.sensor;

public class Sensor5 extends Sensor {

    public int getPort() {
        return 71;
    }
}
```