

# Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies, and Applications

Nick Antonopoulos  
*University of Surrey, UK*

George Exarchakos  
*University of Surrey, UK*

Maozhen Li  
*Brunel University, UK*

Antonio Liotta  
*University of Essex, UK*

Volume II

Information Science  
**REFERENCE**

**INFORMATION SCIENCE REFERENCE**

Hershey • New York

Director of Editorial Content: Kristin Klinger  
Director of Book Publications: Julia Mosemann  
Development Editor: Christine Bufton  
Publishing Assistant: Kurt Smith  
Typesetter: Carole Coulson  
Quality control: Jamie Snavelly  
Cover Design: Lisa Tosheff  
Printed at: Yurchak Printing Inc.

Published in the United States of America by  
Information Science Reference (an imprint of IGI Global)  
701 E. Chocolate Avenue  
Hershey PA 17033  
Tel: 717-533-8845  
Fax: 717-533-8661  
E-mail: [cust@igi-global.com](mailto:cust@igi-global.com)  
Web site: <http://www.igi-global.com/reference>

Copyright © 2010 by IGI Global. All rights reserved. No part of this publication may be reproduced, stored or distributed in any form or by any means, electronic or mechanical, including photocopying, without written permission from the publisher.

Product or company names used in this set are for identification purposes only. Inclusion of the names of the products or companies does not indicate a claim of ownership by IGI Global of the trademark or registered trademark.

#### Library of Congress Cataloging-in-Publication Data

Handbook of research on P2P and grid systems for service-oriented computing : models, methodologies and applications / Nick Antonopoulos ... [et al.].

p. cm.

Includes bibliographical references and index.

Summary: "This book addresses the need for peer-to-peer computing and grid paradigms in delivering efficient service-oriented computing"--Provided by publisher.

ISBN 978-1-61520-686-5 (hardcover) -- ISBN 978-1-61520-687-2 (ebook) 1.

Peer-to-peer architecture (Computer networks)--Handbooks, manuals, etc. 2.

Computational grids (Computer systems)--Handbooks, manuals, etc. 3. Web

services--Handbooks, manuals, etc. 4. Service oriented architecture--

Handbooks, manuals, etc. I. Antonopoulos, Nick.

TK5105.525.H36 2009

004.6'52--dc22

2009046560

#### British Cataloguing in Publication Data

A Cataloguing in Publication record for this book is available from the British Library.

All work contributed to this book is new, previously-unpublished material. The views expressed in this book are those of the authors, but not necessarily of the publisher.

## Chapter 42

# Overlay-Based Middleware for the Pervasive Grid

**Paul Grace**

*University of Lancaster, UK*

**Danny Hughes**

*University of Lancaster, UK*

**Geoff Coulson**

*University of Lancaster, UK*

**Gordon S. Blair**

*University of Lancaster, UK*

**Barry Porter**

*University of Lancaster, UK*

**Francois Taiani**

*University of Lancaster, UK*

### **ABSTRACT**

*Grid computing is becoming increasingly pervasive; sensor networks and mobile devices are now connected with traditional Grid infrastructure to form geographically diverse complex systems. Applications of this type can be classified as the Pervasive Grid. In this chapter we examine how traditional Grid technologies and middleware are inherently unsuited to address the challenges of extreme heterogeneity and fluctuating environmental conditions in these systems. We present Gridkit, a configurable and reconfigurable reflective middleware that leverages overlay networks and dynamic software in response to the requirements of the Pervasive Grid. We also illustrate how Gridkit has been used to deploy a flood monitoring application at a river in the north west of England; this demonstrates both the flexibility Gridkit provides, and how dynamic adaptation optimises performance and resource consumption.*

DOI: 10.4018/978-1-61520-686-5.ch042

## INTRODUCTION

The Grid promises computing as a utility, where distributed computational resources are brought together and can be openly accessed by many via a standardized infra-structure—Grid middleware. Originally concerned with the computational power of networked PCs and cluster computers this has been extended to include pervasively deployed resources embedded within environments. The Pervasive Grid (Davies et al., 2004) merges the vision of ubiquitous computing with traditional Grid computing. The following present some of the many examples of this Pervasive Grid in action:

- **Environmental monitoring and control.** In order to predict natural phenomena such as floods, hurricanes, and volcanic eruptions, scientists collect data and feed this into computationally intensive prediction models. These systems involve networked sensor devices deployed “in the field” that monitor the environment, collect data and then distribute this, via communication networks, to models that may be running local to the monitored site, or running off-site (typically a traditional high-performance Grid).
- **Transport.** Next generation transport systems are embracing pervasive computing. Traffic monitoring systems consisting of sensor devices at the roadside, along with embedded devices within cars collect real-time data that is input to complex traffic models to help improve traffic flow. Similarly the ability for cars to communicate with one another using vehicular ad-hoc networks (VANETS) have been used to improve road safety, by warning drivers or autonomously taking evasive action.
- **Healthcare.** Remote patient monitoring devices e.g. those that are embedded in the home, or mobile devices carried by

the patient monitor their current state of health and are integrated into large-scale healthcare systems to improve standards of patient care. This can include detecting potential problems and informing a suitable healthcare professional.

From these application types it is clear that there is an increasing trend towards *diversity* in Grid applications. Here, two key characteristics of the Pervasive Grid that must be addressed by future middleware are:

1. **Extreme heterogeneity of Grid technologies.** At the *device* level we envisage a spectrum of devices ranging from large cluster computers through to mobile devices, embedded devices and wireless sensors. At the *network* level, the range of network types in use has grown to include: high-speed local networks; lower-speed wide-area networks; infrastructure-based wireless networks; ad-hoc wireless networks and specialised sensor networks. At the *middleware* level, the range of middleware-level communications services in use is expanding from basic point-to-point interactions (e.g. SOAP messaging and RPC), to “interaction paradigms” such as: reliable and unreliable multicast; workflow; media streaming; publish-subscribe; tuple-space/generative communication; and peer-to-peer based resource location or file sharing.
2. **Fluctuating environmental conditions.** Wireless sensor networks deployed on site, mobile computing devices, and ad-hoc networking are all subject to fluctuating conditions in the environment e.g. network quality of service (QoS), resource availability (e.g. battery power), changing location, devices in range, etc.

Dealing with these characteristics is a fundamental challenge for future Grid middleware, and

one that is demonstrably not addressed by existing platforms. We argue that Grid middleware must be flexible and configurable to meet a wide range of application requirements across heterogeneous systems; and importantly should be able to dynamically adapt its behaviour to ensure that it continues to provide the required level of service in the face of changing operating conditions.

In this chapter, we describe Gridkit (Grace et al., 2008) a component-based, *reflective middleware* for the Pervasive Grid. The use of software components as building blocks allows the middleware to be flexibly deployed on heterogeneous devices. Subsequently, reflection is used as a principled approach to adapt these components at run-time. A novel feature of Gridkit is that it leverages *overlay networks* to tackle the problems of network heterogeneity. Overlay networks are virtual communication structures that are logically “laid over” an underlying physical network such as the Internet or a wireless ad-hoc networking environment. The benefits of the overlay approach are that it can mask the heterogeneity of the underlying networked infrastructure, it can provide needed network services (e.g. application level multicast) in network environments that don’t support them, and it is inherently configurable and run-time adaptive so as to be able to address the high degree of dynamicity inherent to pervasive computing environments.

The final part of this chapter examines how Gridkit has been used to deploy a real world application that predicts flooding in a river valley in the North-West of England. In the context of this application, hydrologists deploy sensors (such as depth and flow-rate sensors); and a wireless sensor network is used to: i) feed information into off-site flood prediction models executing upon a traditional computational Grid, and ii) create a “local Grid” of sensors for performing flood prediction computations on site. In this example, we illustrate both the flexibility of middleware deployment across devices, and the use of dynamic adaptation to optimize the performance of the

sensor network and conserve valuable resources such as battery power.

## ADAPTIVE SYSTEMS SOFTWARE

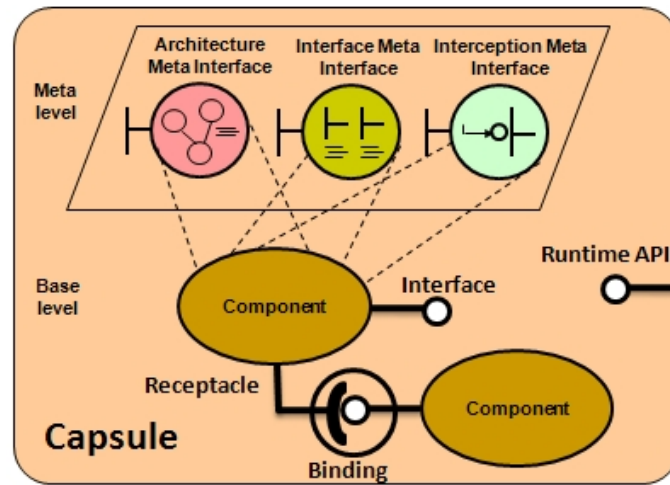
The Pervasive Grid is characterised by change, and as such, next generation Grid software must be capable of dynamically adapting its behaviour in response to this change. In this section we introduce techniques from the field of dynamic software adaptation and in particular focus on the Lancaster philosophy for developing adaptive systems software; this combines three fundamental software technologies: *software components*, *computational reflection* (Maes, 1987), and *component frameworks* (Szyperski, 1998). Here we examine each technology in detail, thus providing the necessary expertise to understand how the Gridkit middleware operates once the three are combined.

### Software Components

Architecture-based runtime software adaptation (OReizy et al., 1998) is a well-established approach to change the behaviour of software over time. Here software components act as the building blocks of adaptive software, and as the unit of change. Inspired by this prior work, **OpenCom** (Coulson et al., 2008) was developed as a general component model designed to support the development of dynamically adaptive systems software (and in particular reflective middleware). An outline of this component model is illustrated in Figure 1.

Here, *components* are language-independent encapsulated units of functionality and deployment that interact with other components exclusively through “interfaces” and “receptacles”. *Interfaces* describe the behaviour offered by each component and are expressed in terms of sets of operation signatures and associated datatypes. Importantly, components support multiple in-

Figure 1. The OpenCom component model



interfaces: this is useful in embodying separations of concern (e.g. between base functionality and component management). *Receptacles* are the corresponding “required” interfaces that are used to explicitly describe the dependencies of one component on other components; *bindings* are then the associations between a single interface and a single receptacle.

Components are deployed and executed within a *Capsule*; these are kernel-like environments that offer a runtime API to manage the contained components. This API provides operations to create component instances, delete component instances, create bindings and destroy bindings. Hence, using this interface components can be deployed and bindings created at any time during run-time (this is called *third-party deployment*). We have developed a wide range of capsules for different devices—e.g. for a Unix or Windows process on a PDA or PC; a RAM chip on a sensor mote; and others; thus allowing software adaptation to take place across heterogeneous pervasive devices.

## Reflection

Reflection is the capability of a system to reason about itself and act upon this information. Hence, a reflective system maintains a representation of

itself that is causally connected to the underlying system that it describes. This is known as the CCSR or Causally Connected Self Representation (Maes, 1987). The CCSR is often referred to as the meta level, and the system itself the base level. Hence, changes made at the meta-level via this self-representation are reflected in the underlying base-level, and vice versa. Reflection is seen as a principled approach to carry out dynamic adaptation.

Figure 1 also illustrates how OpenCom provides reflective behaviour to the component developers. Each traditional OpenCom component (described as a base component) has access to a set of three meta-space behaviours (each with a separate *meta-interface*):

- The *interface* meta-interface supports the inspection of the component’s provided and required interfaces, and the operations available on these interfaces. It also allows third-parties to dynamically invoke methods.
- The *architecture* meta-interface accesses the software architecture of a component (in the following section we will investigate these composite components in further detail). The meta-representation is

a component graph describing the set of connected components, where a connection maps between a required and provided interface in the same address space. Hence, the architecture meta-interface has operations to inspect the current component configuration and also to make changes to this graph structure (which will then be reflected in the base component).

- The two prior meta-interfaces focus on the structure of a system; whereas the *interception* meta-interface allows reflection to manage the behaviour of the system. This interface supports the dynamic insertion of interceptors, these typically describe behaviour to be performed before an operation is executed (a pre interceptor) or after (a post interceptor). Such capabilities provide the ability to add non-functional behaviour implementation into the system at run-time; notable examples would be monitoring or security checks.

## **Component Frameworks**

A Component framework (CF) is a composite component that encapsulates a configuration of components whose role is related to a specific behaviour domain. Hence, this is an abstraction for supporting architectural adaptation as prescribed by (OReizy et al., 1998). In terms of middleware, a component framework could manage a set of network protocol components, service discovery components or buffer management mechanisms. These framework elements also act as the life support environment for the contained components i.e. the framework contains mechanisms to verify valid component configurations (e.g. enforcing a strict stacking architecture for network protocol composition), and it also makes decisions when to perform a dynamic reconfiguration, utilizing appropriate adaptation mechanisms to ensure that these are carried out in a safe and consistent manner. A fundamental requirement of adaptation is

that the system is placed in a safe-state before the adaptation is executed. (Kramer & Magee, 1985) termed this state *Quiescence*; and this involves ensuring that there is no ongoing active computation within the component architecture.

Importantly, component frameworks can either be *node-local* (i.e. the component configuration they manage is within a single address space) or *distributed* (i.e. the component configuration is spread (or replicated) across a number of distinct machines). In the appendix we describe the mechanisms that underpin adaptation for the interested reader. Richer examples of real component frameworks will be presented within the Gridkit middleware.

## **GRIDKIT: RECONFIGURABLE OVERLAY BASED MIDDLEWARE**

The role of middleware is to provide communication services (e.g. resource discovery, remote procedure calls, messaging, and many others) in the face of end-system, operating system and network heterogeneity; and to simplify the task of the application developer by managing the problems of distribution. Established Grid middleware have achieved some success in providing these services; however, in relation to the problems of the Pervasive Grid we believe that they are limited in three respects:

1. Current middleware solutions are '*network-style centric*'. That is, their operation is suited to individual network types, they cannot work in another network type, or across network types. For example, Grid middleware e.g. Globus (Foster et al., 2001) are reliant on the properties of fixed networks such as connectivity. Alternatively, middleware for wireless infra-structure networks or ad-hoc networks typically focus on addressing the problems of network disconnection and mobility.

2. Monolithic implementations of middleware are typically targeted to specific device types (and the majority for high-end desktop PCs or servers). These lack the flexibility to be deployed on the embedded, mobile and sensor devices found in the Pervasive Grid.
3. Middleware technologies typically support individual communication services, e.g. resource discovery, Remote Procedure Call (RPC), publish-subscribe, etc. However, one-size does not fit all network types, and will not optimally work for all types of applications; for example, data sharing is well suited to ad-hoc interaction in ad-hoc and sensor networks, whereas RPC is better suited to infrastructure networks.

One possible solution to these problems is to employ *separate middleware solutions* for different devices, network domains and communication services. This solution is evident in the piece-meal nature of current Grid middleware: e.g. SOAP for messaging, JMS for publish-subscribe, GridFTP for data streaming, and OGSA-DAI for database access. However, this ad-hoc approach has numerous problems:

- The responsibility for middleware composition and integration adds considerable complexity to the application developer’s task.
- The middleware infrastructure becomes redundant and heavyweight due to potentially common functionality being duplicated across multiple implementations (e.g. network transport, resource management, and security).
- Individual middleware implementations may only operate in certain environments and/or under certain network conditions (e.g. separate publish-subscribe implementations are typically used for infrastructure-based and for ad-hoc networks)—this again leads to redundant deployment, this time of whole middleware services.

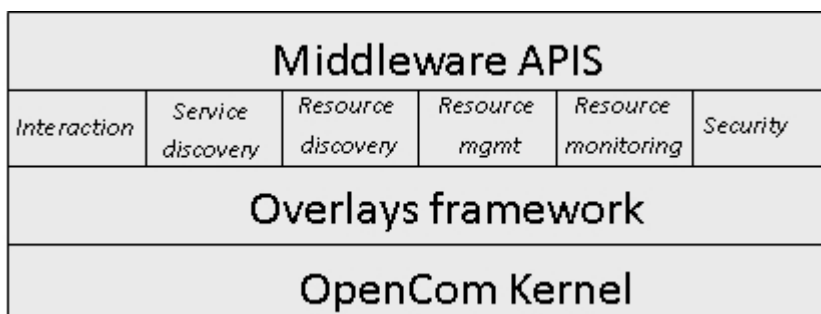
Therefore, Gridkit has been designed to provide a flexible range of middleware services across a range of network domains and device types. This common software framework aims to transparently hide the complexities of heterogeneity and middleware configuration from the application developer and also provide consistent programming APIs for communication services. By achieving these requirements Gridkit is a natural tool to support the development of Pervasive Grid applications. In this section we focus on the overall architecture of Gridkit describing its core component framework and how configurations are generated.

## The Gridkit Architecture

An illustration of the high-level software architecture of Gridkit is provided in Figure 2. The framework is built from OpenCom components and executes within the OpenCom kernel represented at the bottom of the diagram. Each of the boxes above describe an individual component framework related to a particular domain of Gridkit’s functionality; these illustrate a central philosophy of capturing domain behaviour within individual frameworks in order for them to be developed and optimized individually by domain experts. Briefly, the *overlays framework* is a distributed framework for the deployment of multiple overlay networks. In practice, this amounts to hosting, in a set of distributed overlay framework instances, a set of per-overlay plug-in components. This framework provides a virtualized view of network behaviour (in potentially many different environments) to allow higher level services and frameworks to be easily deployed without being concerned about the underlying network heterogeneity.

Above the overlays framework is a set of further “vertical” frameworks that provide functionality in various orthogonal areas, and can optionally be included or not included on different devices. The frameworks are as follows: the *interaction framework* accepts multiple interaction type plug-ins (e.g. RPC, publish-subscribe, group

Figure 2. The Gridkit architecture



communication); the *service discovery framework* accepts plug-in strategies to discover application services (e.g. SLP, UPnP, Salutation); the *resource discovery* framework accepts plug-in strategies to discover resources such as CPUs and storage (e.g. peer-to-peer search); the *resource management* and *resource monitoring* frameworks are respectively responsible for managing and monitoring resources; and the *security* framework provides general security services for the rest of the frameworks.

### The Overlays Framework

The overlays framework (as visualised in Figure 3) is an OpenCom component framework that is deployed on each participating node in the distributed system. The framework accepts ‘plug-in’ components that offer various types of overlay-related behaviour. More specifically, the types of components that can be plugged into the framework are as follows.

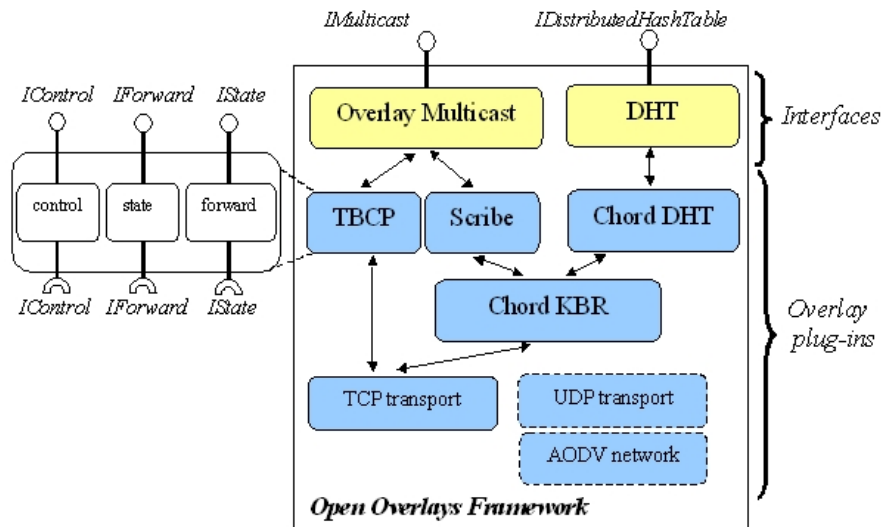
*Overlay plug-ins.* These are per-node implementations of network overlays. For example, Figure 3 shows four overlay plug-ins: TBCP (Mathy et al., 2001) which is an Internet Scale application level multicast protocol; the Chord Key-Based Routing (KBR) overlay (Stoica et al., 2001) is a lookup protocol to find the nearest host to a given identifier in a ring-structured overlay; Scribe (Castro et al., 2002) is a multicast protocol layered directly upon key-based routing

behaviour; and the Chord Distributed Hash Table (DHT) is a decentralised service similar to a hash table, where data is stored and retrieved from the overlay node closest to the hash of the data’s key.

*Interface plug-ins.* While overlay plug-ins provide different types of behaviour, interface plug-ins capture common API patterns that can be shared by multiple overlays. We have used the key abstractions described in (Dabek et al., 2003) as the basis of these APIS, as these are generally considered the de-facto standard for overlay interfaces. Figure 3 shows interface plug-ins for DHT and multicast overlays. The indirection provided by interface plug-ins isolates higher-layer software from the idiosyncrasies of individual overlay plug-ins, facilitates application-transparent adaptation, and encourages a principled approach to the development of ‘families’ of overlays plug-ins, each of which shares a common API.

One of the key features of the framework is that multiple overlays can operate simultaneously in the framework either in mutual isolation (cf. TBCP in Figure 3) or in a stacking relationship (e.g. Scribe and Chord DHT are both stacked atop Chord KBR). Hence, multiple types of overlay behaviour can support a wide range of middleware services, and re-using lower-level plug-ins reduces the duplication of networking implementation. The overlay plug-in abstraction is applied uniformly throughout the communication stack. For example, transport protocols like

Figure 3. An example configuration of the open overlays framework



TCP or UDP are represented as overlay plug-ins (as seen at the bottom of Figure 3); these could be replaced by an AODV overlay plug-in at the network layer should the device be operating in a MANET environment.

The overlay plug-ins are themselves coarse-grained modules; hence, we also propose a software pattern for developing these plug-ins that promotes configuration, reconfiguration, and re-use at a finer-grained level within each overlay. Each Overlay plug-in is a ‘mini’ component framework, each of which, as shown in the left part of Figure 3, is composed of three distinct components that respectively encapsulate the following areas of behaviour:

- i. **Control behaviour**, in which the node co-operates with its peer control element on other nodes to build and maintain an overlay-specific virtual network topology;
- ii. **Forwarding behaviour** that determines how the overlay will route messages over the aforementioned virtual topology;
- iii. **State** information that is maintained for the overlay; e.g. nearest neighbours.

Each of these three elements exposes a standard interface, IControl, IForward, and IState respectively, which enables the free composition of overlays (subject to the configuration constraints discussed below). We refer to this three-element architecture as the overlay pattern. The motivation for the overlay pattern is to achieve flexibility in terms of both configuration and dynamic reconfiguration by enabling both control and forwarding behaviour to be independently replaced without loss of state information. We see an example of this fine-grained adaptation in the flood monitoring application discussed later.

### Benefits of the Overlay Framework

In terms of the effectiveness, the overlays framework provides benefits in three key dimensions (note we leave the demonstration of reconfiguration benefits until later).

**Generality.** Table 1 lists the network types that have been developed for and deployed within the overlay framework. From this list, it is clear that the framework can provide a general range of network services across a range of heterogeneous environmental conditions. There are different

*Table 1. Descriptions of some implemented overlay plug-ins*

Overlay Name	Description and configurability options
Chord KBR	A large-scale KBR overlay based on the Chord algorithms (Stoica et al., 2001).
DHT	A storage overlay for a decentralised lookup of large amounts of data (equivalent of hash table behaviour).
Pastry KBR	A KBR overlay based on Pastry (Rowstron et al., 2001).
Failure Monitor	Monitoring overlay based upon the algorithms described in (vanRennesse et al., 1998); this detects node failures and notifies other members of the overlay.
SCAMP	A scalable group membership overlay with gossip-based forwarding as described by (Ganesh et al., 2001).
Scribe	A multicast overlay that can be layered on top of any KBR overlay (e.g. Chord or Pastry) as described by (Castro et al., 2002). This is well suited to large-scale and highly dynamic multicast memberships.
Spanning Trees	A tree structured overlay for fan-in routing i.e. routing data from individual nodes to the root of the tree. These are well suited to static ad-hoc network deployments (e.g. networked sensors). These can be configured to different tree topologies e.g. shortest hop trees or fewest hop trees.
TBCP	A wide area multicast overlay for Internet scale multicast membership with a low degree of membership churn as described by (Mathy et al., 2001).

types of KBR protocols (e.g. Chord and Pastry), a DHT overlay, multicast protocols (e.g. Scribe and TBCP), gossip overlays (Scamp), and more specialised overlays such as a node failure monitoring overlay, and a spanning tree overlay for fan-in routing in network sensors. Further more in terms of providing *virtualisation* of network services to higher-level middleware services Table 1 also illustrates the range of overlays that can be virtualised by common interfaces, e.g. multicast and DHT network resources.

**Configurability.** To measure the configurability of the framework we calculated the number of possible configurations in each of four profiles (i.e. an ‘empty’ profile consisting of only the framework itself, a ‘WSN’ profile for wireless sensor network environments containing only the spanning tree overlays, a ‘multicast’ profile for all the multicast overlays from Table 2, and a ‘full’ profile containing all 8 overlays. The numbers, which are summarised in the rightmost column of Table 2, result from an exhaustive enumeration of all the configurations reachable. The results show that the more complex and well-populated profiles support a very large number of possible configurations; e.g. the ‘full’ profile has 26,999. Furthermore, the overlay pattern contributes significantly to the configurability of the framework by supporting

*Table 2. Configurability results and overheads for framework profiles*

Profile	Static Memory cost of Overlay plug-ins (KB)	Total configurations
Empty	60	1
WSN	146	4
Multicast	169	89
Full	252	26,999

fine-grained configuration of individual overlays. Consider, for example, a Gnutella implementation with either a random-walk-based, or a flooding-based forwarder; or a tree overlay with a control element that either contains or doesn’t contain a self-repair algorithm.

**Resource overhead.** What is the resource overhead incurred by the overlays framework when providing such configurability and generality? We performed three experiments to determine this.

These employed components executing on a Java 1.5.0.10 virtual machine on a networked workstation with a 3.0 GHz Pentium 4 processor, 1 Gbyte of RAM and running Windows XP. The first experiment (see Table 2) investigated the *static storage footprint* costs; i.e. the disk space required to store the framework, and components. It can be seen that the base framework requires

60K before any plug-ins are added. In the second experiment (see Table 3), we evaluated *dynamic memory overhead* by measuring the RAM footprint of overlay plug-ins while they were in operation (i.e. joined to a running overlay). We can see that the basic framework with no plug-ins is responsible for a high percentage of the overall footprint (65% on average; note, however, that this figure includes 6,392 Kbytes for the JVM and 600 Kbytes for the OpenCOM kernel). We can again reduce overhead in a given deployment through the profiling mechanism. More complex configurations, e.g. the layering of Scribe over a Chord KBR obviously increases the footprint size, but by a small margin, e.g. adding Scamp to TBCP results in a 164 Kbytes increase.

Finally, the third experiment (again, see Table 3) investigated *configuration performance* by measuring the time needed to configure new plug-ins based on a sample of configurations from the different profiles (e.g. TBCP in the full profile, etc.). While it is clear that configuration performance is largely tied to the complexity of the configuration in terms of the numbers of configuration rules and plug-ins involved, and the number of inter-component connections etc., the overall cost of configuration is largely negligible compared to time for a node to join an overlay (e.g. Pastry averages 5 to 10 seconds for node joins).

## The Interaction Framework

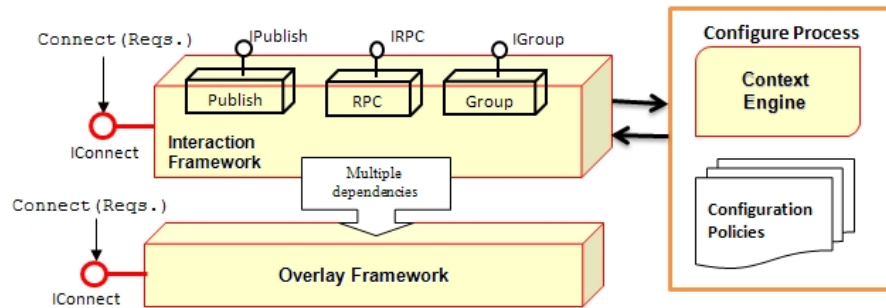
Gridkit’s interaction framework provides a common framework for an extensible set of so-called *pluggable interaction paradigms*, or *PIPs*. The overall architecture and context of the interaction framework is illustrated in Figure 4. Here individual PIPs are component frameworks implementing a particular type of communication service. For example, an RPC PIP will typically have components that: marshal and unmarshal RPC messages, forward these to appropriate local objects, and manage object lifetimes; whereas a publish-subscribe PIP will contain components to: parse/construct event messages, maintain subscription information, and match events against subscriptions. These are implemented with the philosophy of separating the interaction framework from the overlay framework to promote the reuse of overlays and thus conserve resources—i.e. different interactions may re-use overlay configurations that are already in place (for example, a topic-based publish-subscribe PIP and a reliable multicast PIP might both share a multicast tree overlay). The design of the interaction framework is guided by the following principles:

1. The selection and use of PIPs by applications should be straightforward;
2. The programming model of each PIP should be independent of how it is implemented

Table 3. Performance times and dynamic memory costs of typical configurations

Configuration	#Components	#Bindings	Profile	Configuration time (ms)	Dynamic memory (KB)
Empty	0	0	Full	N/A	10,448
Empty	0	0	Sensor	N/A	8,352
Spanning tree	5	12	Sensor	191	11,452
TBCP	6	12	Full	211	15,144
SCAMP	5	9	Full	152	13,708
Scribe/KBR	9	27	Full	486	16,652
Scribe + TBCP	13	39	Full	592	16,972
TBCP+SCAMP	10	21	Full	281	15,308

Figure 4. The interaction framework



over different (overlay) network types and conditions;

In line with the principles above, we have made every effort to simplify the API of the interaction framework. General experience in the development of reflective middleware has taught us that highly configurable systems are often a two edged sword: configurability is certainly a good thing, but too often the benefits are outweighed by the inconvenience and complexity of having to write many lines of baroque code to achieve a desired configuration. In many cases, this complexity is so great that developers ignore the available flexibility and use only a small number of default configurations. This is especially relevant in the case of the interaction framework as (unlike the overlay framework) it is generally used directly by application developers.

To illustrate the abstractions provided to application developers we give an example of a generic API. In the *IPublish* API for publish-subscribe PIPs (see below), *createChannel()* creates an event dissemination channel or network. This can be an individual channel for all events (e.g. a broker network) or a specific topic channel (implemented by an individual group). *Publish()* disseminates an event on the channel. Apart from the primary content, additional information (an example could be the location of the publisher) can be passed using the *Context* parameter. *Subscribe()* subscribes to a particular channel and takes a

content/context filter (subscriptions are defined in a dedicated filter language, and an *EventListener* as parameters. The event listener will respond to the *matchedEvent* notification. From this interface, the transparency from configuration and overlay heterogeneity is clear.

```

TopicID =
createChannel (UserDefinedID);
joinChannel (ChannelID);
publish (TopicID, Content, Context);
subscribe (TopicID, Filter, EventListener);
Notification -> matchedEvent;
    
```

### Configuring Middleware Behaviour

We now describe how middleware behaviour is configured in Gridkit. Given a request for a particular PIP by the application developer, we show how the interaction framework and overlay framework are configured. Our approach here employs a notion of so-called *binding contracts*. More specifically, PIP interfaces have attached to them sets of *name-value pairs* that embody PIP-specific information such as the name of the PIP, its purpose, constraints on its use, and the QoS it provides. Correspondingly, the receptacles of application components that want to use PIPs have *predicates* attached to them whose terms refer to the name-value pairs attached to potentially match-

ing PIP interface references. The binding contract elements (i.e. name-value pairs and predicates) are attached to receptacles and interface references using native facilities of our component model (i.e. the ‘interface’ meta-interface). This process repeats throughout the Gridkit architecture i.e. the interaction framework requests behaviour from the overlay framework using the same approach and configuration mechanism.

Based on binding contracts, we provide a simple generic API to the interaction framework of the form *connect(receptacle)* to which the potential user of a PIP submits its receptacle. This is illustrated to the left of Figure 4. Given this, the interaction framework selects, instantiates, and configures a PIP instance based on the following information:

- The set of PIPs currently registered with the interaction framework;
- The predicates attached to the offered receptacle;
- The advice of a *context engine* which follows the design as proposed by (Capra et al., 2003) which supports additional name-value pairs the value of which varies dynamically according to the context of the host machine (e.g. battery life, network connectivity etc.).
- *Declarative configuration rules* are XML-based expressions that specify the

configuration possibilities supported by the profile. As an example, a configuration rule could state that when a ‘multicast’ service is requested by the application, and the current network context is ‘fixed infrastructure with no IP multicast support’, then the TBCP overlay plug-in should be instantiated and configured beneath the ‘overlay multicast’ interface plug-in.

Consider a Gridkit installation that is described by Table 4. This shows the plug-ins that are currently registered with the interaction and overlay frameworks, and the context on each of the two device types, namely a PC or a PDA. It also shows the current set of name-value pairs for the plug-ins and the per-device context. *RelMsg* means reliable messaging; *GrpMem* means group membership; and *Net* means network type (i.e. either fixed infrastructure or ad-hoc networking). Given this installation, consider the processing of a request on the interaction framework of the form *connect(publish\_receptacle)* for a Publish PIP where there is a predicate of the form *RelMes=F* attached to *publish\_receptacle*. The steps involved in processing this request are as follows:

- **Step 1:** The *connect()* call is issued on the interaction framework as already described.
- **Step 2:** The interaction framework retrieves from the engine, context relevant

Table 4. An example Gridkit installation

Framework	Generic API	Item	Name-value pairs
Interaction	<i>IPublish</i>	Publish	RelMes: F
	<i>IGroup</i>	Group1	RelMes: F; GrpMem: T
		Group2	RelMes: F; GrpMem: F
Overlay	<i>IGroupMessage</i>	ALM	RelMes: F; Net: fixed
	<i>IGroupMessage</i>	ProbMcast	RelMes: F; Net: adhoc
	<i>IGroupMembers</i>	Gossip	RelMes: F; Net: fixed; Net: adhoc
Context	N/A	PC	Net: fixed
		PDA	Net: adhoc

to the type of PIP being requested (in this case *Net: fixed* for PC, or *Net: adhoc* for PDA).

- **Step 3:** A pattern matching algorithm fires the appropriate configuration policy, instantiates the PIP and then decides on a suitable overlay to underpin the PIP.
- **Step 4:** The script issues a *connect(alm\_receptacle)* call on the overlay framework and the process is repeated.

For a request for a *Group* PIP with a receptacle predicate of *RelMes=F* and *GrpMem=T*. A similar process to the above will be carried out with the *Group1* PIP being selected (because of the specification of *GrpMem=T*), and underpinned by *ALM* and *Gossip* overlays on the PC, and *ProbMcast* and *Gossip* overlays on the PDA (due, as above, to the contextual differences). The *Gossip* overlay is used to gossip about group membership (as required by the *GrpMem=T* part of the predicate).

Finally, the interaction framework also supports *dynamic monitoring* of currently instantiated binding contracts. Using this facility, either party to the binding contract (including the context engine) can force a re-evaluation of the contract by altering their respective 'side' of the contract. For example, the user of the framework can drive reconfiguration of a PIP by altering the predicates attached to its receptacle. To detect such changes, the component model's 'interception' meta-interface is used to attach a 'dynamic contract evaluator' to the receptacle-interface binding, which is executed every time a call is made across the binding, and raises an exception if it finds the binding contract to be no longer valid. This exception can either be handled by the user of the framework or by the framework itself, e.g., delete the PIP instance or attempt to reconfigure it. As an example, the context engine might change a name-value pair to reflect the fact that a live Ethernet MAC layer no longer exists, and the framework might on that basis change the underlying overlay from

IP-based multicast routing to an ad-hoc network based multicast approach.

## GRIDKIT IN A FLOOD MONITORING APPLICATION

We now describe how Gridkit has been used to develop a real-world Pervasive Grid scenario: namely a wireless sensor network-based real-time flood forecasting in a river valley in the north west of England. In this scenario, a wireless sensor network (WSN) comprising of 20 nodes has been deployed to monitor depth and flow conditions along a 2.5KM stretch of river. We discuss the hardware, software and networking solutions that were chosen and then deployed on site. Subsequently we discuss the dynamic nature of the environment and present how dynamic reconfiguration of the overlay framework is used to improve both the performance and resource consumption of the application.

### The GridStix Sensor Devices

Each sensor node (known as 'GridStix') comprises a 400MHz XScale CPU, 64MB of RAM and 16MB of flash memory. They have the capability to communicate via three types of networks: i) 802.11b/g for high performance long range networking, ii) Bluetooth for low power short range networking, and iii) GPRS/UMTS to connect to the Internet. Each GridStix is powered by a 4 watt solar array and a 12V 10Ah battery. They run Linux 2.6, version 1.4 of the JamVM Java virtual machine. Finally, they provide options to connect a number of different sensors:

- Pressure-based depth sensors that connect via ADC Channels.
- Ultrasound-based flow monitoring that connect via RS-232.
- A Digital camera for image-based flow measurement that connect via Ethernet.

The devices themselves are shown in Figure 5; this illustrates the combined package of sensor board, and power supply; and how they are deployed in the river. Here a sensor in the water is connected directly to the GridStix device. Note, that the device is relatively large, as in such a scenario there is no requirement to minimise the size of the sensor.

## GridStix Software Configurations

The flood monitoring system monitors water depth (using pressure sensors) and flow-rate (using a combination of image-based flow measurement and ultrasound flow measurement). This data is then fed into flood models which then predict when flooding is likely to occur, before notifying interested stakeholders of these events. In this application the following requirements of Gridkit are important:

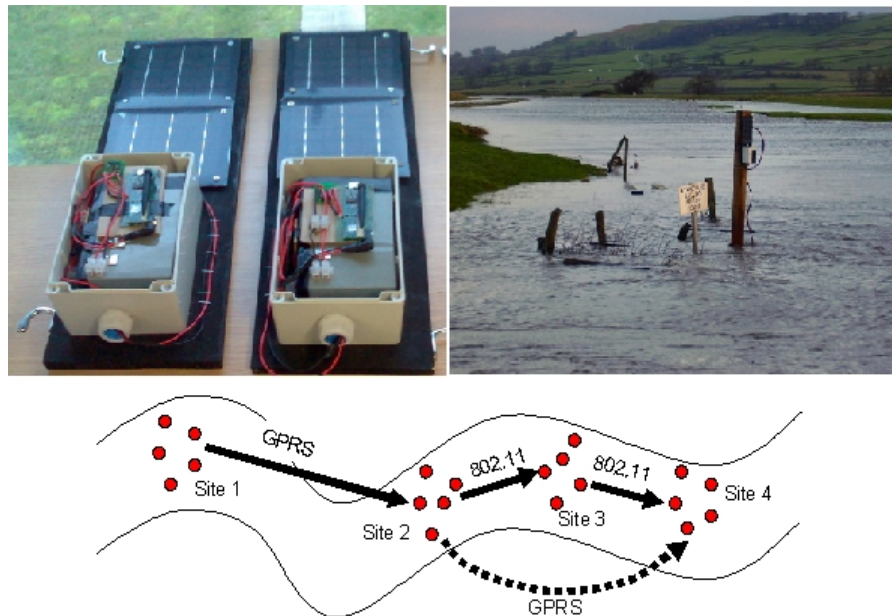
1. **To perform flood prediction using spatial flood models.** This must collect data from all sensor nodes, forward this data to the root nodes of the sensor network whose role is to then distribute data off-site (using GPRS/UMTS) where the models execute. Notably, these models are computationally complex and require traditional cluster or Grid computing solutions.
2. **To perform site-local flood prediction using Point-prediction models (Beven et al., 2005).** Here, data is collected from the sensors and then forwarded to root nodes who manage the execution of the models on-site because there is significantly less computational overhead compared to spatial models.
3. **To use image-based flow measurement to help flood prediction.** Digital imaging sensors (e.g. a digital camera) collect data sets that cannot easily be transmitted off-site (without significant expense due to their

size). Hence, flow measurement is performed at the site; however, the computational costs prohibits single sensor execution, hence the task must be distributed across the local sensors (c.f. a local mini-grid).

Figure 5 also illustrates the location of the deployed sensor network. The sensors are placed at locations along the river; depending on the distances between sensors, different networks are used to communicate e.g. Bluetooth and 802.11b for shorter distances, GPRS for longer communication. To meet the communication requirements of flood prediction application described above, we deploy the Gridkit wireless sensor network profile. In the interaction framework the publish-subscribe PIP is employed; a node can be just a publisher (e.g. a child node), just a subscriber, or, optionally, it can act as a broker in the event service (e.g. a root node). Underpinning this framework is a set of overlays for the distribution of events towards sinks. This is also customizable from: i) a centralized spanning tree using Dijkstra's shortest path algorithm, or ii) a fewest hop tree. *Fewest hop* (FH) spanning trees are optimised to maintain a minimum number of hops between any given node and the root. FH trees minimise the data loss that occurs due to node failure, but are sub-optimal with respect to power consumption. *Shortest path* (SP) spanning trees are optimised to maintain a minimum distance in edge weights from any given node to the distinguished 'root' node; edge weights are derived from the power consumption of each pair-wise network link. SP trees tend to consume less power than FW trees, but offer poorer performance;

Finally, at the physical network level (i.e. the bottom level of the overlays framework) each node can use either *Bluetooth* or *WiFi* (802.11b). Both technologies have extremely different throughput, energy, and range properties. WiFi provides the highest throughput and longest range, but a cost of energy consumption almost an order of mag-

Figure 5. The GridStix sensors deployed at the river



nitude higher than Bluetooth. Typically Bluetooth would be used in quiescent conditions, and WiFi in imminent flooding situations.

### Dynamic Reconfiguration

This Pervasive Grid application necessitates rich support for heterogeneous network technologies. On the one hand, networking support must be sufficiently power-efficient that nodes may operate for extended periods of time. On the other hand, applications such as image-based flow prediction also require high performing (and implicitly power hungry) networking support. This need for heterogeneity is further compounded by varying resilience requirements: During quiescent periods, when flooding is unlikely, data may reach the off-site cluster with a high delay. Faults in the network may take a long time to be recovered from, since they might only jeopardise the completeness of measurement logs. In these periods, a low energy consumption is a prime requirement to maximise the life-time of the sensor network. By contrast, when a flood is imminent, we want the network

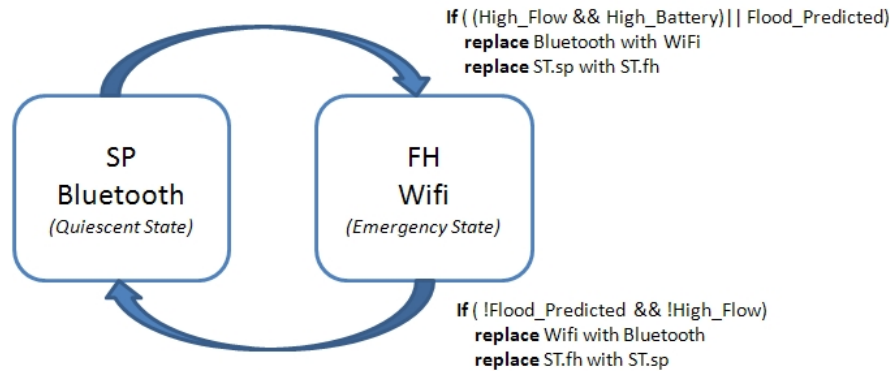
to react quickly, while providing a high degree of resilience (e.g. a low sensitivity to disruptions), even if this means its energy supplies get depleted much more rapidly. Hence, we examine how dynamic reconfiguration of the overlays framework can meet these demands.

### Triggering Reconfiguration

Reconfiguration is supported in our sensor network through the Distributed Component Framework facility included in the Overlays Framework. The reconfiguration ‘triggers’ that drive the system from one configuration to another are expressed with declarative configuration rules, summarised in Figure 6 in the form of a state transition diagram (to avoid excessive presentational complexity, the diagram represents a drastically simplified view of the implemented system i.e. only two of the possible states of the system). We also show the pseudo-code rule relating to each of the transitions.

The triggers/rules are based upon optimising configurations for three factors: latency, resilience and power consumption in different environmen-

Figure 6. Reconfiguration states and triggers (simplified)



tal contexts. To determine these configurations a number of experiments were performed on the real sensor network deployment. These experiments are described in full detail in (Grace et al., 2008) and describe how the states are chosen. There are three important context types at the base of reconfiguration. First, *High\_Flow* is a context that is detected by attaching a video camera to some of the nodes, pointing this at the river surface, and estimating river flow rates by carrying out some simple image processing on the resultant images. Secondly, *Flood\_Predicted* is provided by point prediction models which provide localised predictions of water depth based on the collated readings of depth sensors in the immediate locality. Finally, *High\_Battery* is measured by collecting the current power readings of individual sensor nodes. For a simplified explanation: we can see in the diagram that when flooding is predicted we switch to the more resilient fewest hop tree that consumes more power, but in a quiescent state we switch to the shortest path tree to conserve resources.

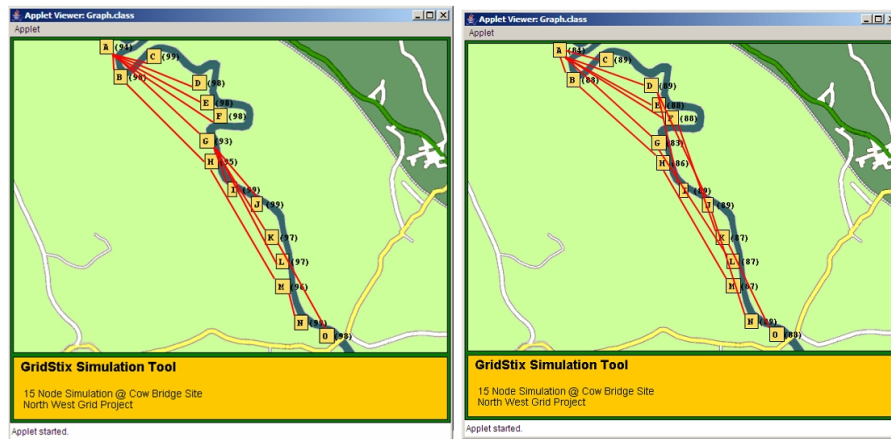
### The Benefits of Reconfiguration

We evaluated reconfiguration primarily through simulation of the Gridkit middleware in operation in an example sensor deployment. This is because it provided the control to simulate events that trigger

reconfiguration. For this, we developed the *GridStix simulator* which models the low-level properties of each node (available CPU, available Battery, solar panel power production) and each pairwise network link (round-trip-time, power-consumption, bandwidth, delay, jitter, loss). This low-level data has been measured empirically on the real-world system, which makes the simulator highly accurate for this scenario. The visualisation sub-system of the simulator is shown in Figure 7 illustrating FH (left) and SP (right) overlay configurations.

The simulation was configured as follows: The simulation period is 24 hours (midnight-to-midnight). Each node enters the simulated period with a battery at 50% charge. Flood conditions begin at 12PM and last until 6PM (the approximate mean duration of a flood event at the site). Dawn occurs at 8AM from which time solar power production is set to WINTER\_SUN, when flood conditions begins at 12PM, solar power production is set to HEAVY\_CLOUD, finally night falls at 8PM (approximately mimicking late winter conditions, when flooding is most prevalent). All nodes were programmed to wake for one minute in every hour. During quiescent conditions nodes transmit sensor readings at a rate of one per minute. During flooding condition, nodes transmit sensor readings at a rate of one per second. This is commensurate with the increased requirements of performing real-time flood modelling.

Figure 7. FH (left) and SP (right) Spanning Trees in the Gridstix simulator



Throughout the simulated period, the system's performance has been evaluated in the context of three key metrics:

1. **Resilience:** The resilience of the network is a function of the extent to which the failure of a given node reduces the overall connectedness of the network. We measure this as the number of viable routes between each node and the root.
2. **Power consumption:** Although the GridStix are equipped with solar panels, power consumption is still an extremely important factor. We infer this from the per-node battery power consumed throughout the test.
3. **Performance:** We measure this in terms of the latency with which messages can be relayed from each sensor node to the root node.

We measured in the simulation for three types of overlays: i) an SP configuration, ii) a FH configuration, and iii) the adaptive configuration defined by Figure 6. In the simulation the flooding is detected (at noon), and the middleware reconfigures from using a low-power SP tree to a high-performance FH tree. We found the following key results:

Resilience

- As FH trees have a typically lower node degree they tend to be more resilient to node failure.
- SP trees have a typically higher degree and are therefore significantly more vulnerable to node failure.
- Where nodes reconfigure between overlays during flooding, system resilience matches SP during quiescent periods and FH during flooding periods (i.e. it is more resilient, but at the expense of power).

Battery Power

- SP trees maintain the highest possible battery life throughout the test.
- FH trees result in the greatest battery power consumption (though at the expense of performance and resilience as shown).
- Where the system reconfigures at flood time from an SP to a FH configuration, battery power is maximised during quiescent conditions (i.e. approximating SP), while increasing during flooding conditions (i.e. approximating FH); though at the same time providing better performance and resilience.
- The power consumed by reconfiguration, both in the transmission of reconfiguration

messages and in CPU-usage is acceptably low.

#### Performance

- FH configurations demonstrate a mean reporting latency of 11ms, while SP offers a reporting latency of 28ms (though consuming significantly less power).
- When the system reconfigures from an SP configuration during quiescent conditions to an FH configuration during flood conditions, performance is correspondingly low during quiescent conditions, but high during flood conditions (though at the expense of power).

In summary, testing on a real-world system illustrates the benefits of overlay reconfiguration. By configuring to a low-power overlay during normal conditions and a high performing and resilient overlay during flood conditions, battery life is extended, while maintaining system functionality during critical conditions. We also found that the cost of reconfiguration in terms of additional power consumed, and reconfiguration time does not significantly affect the power costs and performance of the sensor network.

## FUTURE TRENDS FOR GRID MIDDLEWARE

The implementation approach currently favoured by Grid middleware (e.g. OGSA) is to layer the Grid environment on top of existing web services platforms. Although these platforms are a useful starting point, they have significant limitations as a Grid middleware support infrastructure for the next generation of applications required within the Pervasive Grid. Here we examine how Grid middleware can follow the trends from the wider middleware community, and also the lessons learned from configurable and reconfigurable

middleware (like proposed in this chapter).

*Firstly*, Grid middleware is extremely limited, in comparison to object-based middleware platforms (e.g. RM-ODP and CORBA, and industry-developed platforms like Java RMI, Enterprise JavaBeans, DCOM, and the .NET remoting architecture) in the following areas:

- **The provision of generic (horizontal, or breadth-oriented) services:** For example, CORBA supports generic reusable services like fault tolerance, persistent state, automated logging, load-balancing, transactional object invocation, event distribution, and many others;
- **Scalability and performance:** For example, EJB and the CORBA Component Model have sophisticated support for the automated activation/ passivation of stateful services, and natively support services that span multiple machines/ networks; in addition performance engineering has been the subject of intensive research in the object-based middleware community over the last 10 years.

In contrast, horizontal services are conspicuously lacking in current Grid environments and there is great potential here for reuse from the wider field. And in terms of performance, the application focus of web services-derived middleware has traditionally been on e-Commerce where dependability and security are far more important than performance.

*Secondly*, web services-derived platforms have little or no support for QoS specification and realisation. We believe that such facilities will be increasingly demanded by sophisticated pervasive applications. We also believe that a prime cause of this deficiency is an over-reliance by web services platforms on SOAP as a communications engine. Although very flexible and general, SOAP clearly shows its limitations when relied on exclusively:

- It is inappropriate for Grid applications involving large-volume scientific datasets mainly due to its use of XML as an on-the-wire data representation. This is highly demanding in terms of bandwidth, memory and processing cycles (especially compared to earlier standards like CORBA's CDR). Alternatively it also heavyweight when considering the embedded and sensor devices discussed in this chapter.
- Although it offers some flexibility in terms of support for different interaction types (e.g., choice of request-reply or one-way messages), it does not support the comprehensive range of interaction types proposed within this chapter.

The OGSA design recognises the limitations of exclusive reliance on SOAP, and leaves room for non-SOAP bindings (e.g. using CORBA IIOP and, potentially, other bindings that do have some support for QoS). However, OGSA does not currently specify any particular framework whereby such bindings can be integrated into the distributed programming model, and it similarly does not provide any framework for generic QoS specification/ enforcement.

*Thirdly* and finally, Grid middleware does not make use of advanced middleware research which is investigating highly configurable (and run-time reconfigurable) reflective and component-based middleware technologies. A prime, and highly successful, example of such a platform is the open source JBoss application server (Fleury & Reverbel, 2003). The basic philosophy of advanced middleware (c.f. Gridkit) is to support configurability, extensibility and adaptability as fundamental system properties. In particular, the approach enables alternative policies (e.g. security policies, replication policies, service (de)activation policies, priority-assigned invocation paths, thread scheduling) and components (e.g. protocols, buffer managers, loggers, debuggers, demultiplexers) to be configured in or out at deploy-time,

and reconfigured at run-time (e.g. on the basis of dynamically evolving conditions).

Overall, our view is that next generation Grid middleware can and should leverage the results of the wider middleware field as discussed above. In doing so, it can retain key web services-derived characteristics (loose coupling, XML-based data structuring, reliance only on ubiquitous Internet standards) while additionally folding in some of the key benefits of the wider field—in particular, the availability of generic services, and scalability and performance engineering know-how offered by 'standard' middleware; and the increased flexibility and configurability promised by advanced middleware research.

## **CONCLUSION**

In this chapter we have presented the case for novel Grid middleware solutions to tackle the problems of heterogeneity and dynamic change in Pervasive Grid applications. We have documented how the Gridkit middleware addresses some of the many challenges; in particular how two complementary component frameworks support an extensible set of interaction paradigms and an extensible set of overlay networks. The combination of the two frameworks enables a wide range of pluggable interaction paradigms to be instantiated in a wide range of network environments. Notably, we have also presented a real world flood monitoring application that illustrates the effectiveness of the middleware in both supporting the developer deploy a working solution, and also in optimising the performance of the system in the face of changing conditions. We illustrated how the co-ordinated reconfiguration of middleware components across a sensor network deployment provided performance gains, resilience gains, and improved battery life.

Based upon these results we are particularly interested in supporting challenging scenarios involving 'extreme' network heterogeneity e.g.

involving systems that span a sensor network, a fixed grid environment, and a loosely-connected MANET. This is a fundamentally challenging issue in that it is not yet understood even how to design overlays that can successfully span such environments, let alone an overarching framework. In addressing this challenge, we do not foresee major problems in applying the basic tenets of our framework on individual nodes; it will be the distributed deployment and reconfiguration issues involving distributed adaptation that will present the major challenges (e.g. making appropriate choices in terms of distributed versus centralised configurators, quiescence and validation algorithms, membership protocols, etc.).

## REFERENCES

- Beven, K., Romanowicz, R., Pappenberger, F., Young, P., & Werner, M. (2005). The Uncertainty Cascade in Flood Forecasting. In *Proceedings of the ACTIF meeting on Flood Risk*, Tromsø.
- Capra, L., Emmerich, W., & Mascolo, C. (2003). CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29(10), 929–945. doi:10.1109/TSE.2003.1237173
- Castro, M., Druschel, P., Kermarrec, A.-M., & Rowstron, A. (2002). SCRIBE: A Large-Scale and Decentralised Application-Level Multicast Infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8), 1489–1499. doi:10.1109/JSAC.2002.803069
- Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J., & Sivaharan, T. (2008). A Generic Component Model for Building Systems Software. *ACM Transactions on Computer Systems*, 27(1), 1–42. doi:10.1145/1328671.1328672
- Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., & Stoica, I. (2003). Towards a Common API for Structured P2P Overlays. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*.
- Davies, N., Friday, A., & Storz, O. (2004). Exploring the Grid's Potential for Ubiquitous Computing. *IEEE Pervasive Computing / IEEE Computer Society [and] IEEE Communications Society*, 3(2), 74–75. doi:10.1109/MPRV.2004.1316823
- Fleury, M., & Reverbel, F. (2003). The JBoss Extensible Server. In *Proceedings of the IFIP/ACM Middleware Conference*, (LNCS, pp. 344-354). Berlin: Springer Verlag.
- Foster, I., Kesselman, C., & Tuecke, S. (2001). The Anatomy of the Grid: Enabling Virtual Organizations. *The International Journal of Supercomputer Applications*, 15(3).
- Ganesh, A., Kermarrec, A., & Massoulié, L. (2001). SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Proceedings of the 3rd International Workshop on Networked Group Communication*.
- Grace, P., Hughes, D., Porter, B., Coulson, G., Blair, G., & Taiani, F. (2008). Experiences with Open Overlays: A Middleware Approach to Network Heterogeneity. In *Proceedings of the 3rd ACM International EuroSys Conference* (123-136). New York: ACM Press.
- Kon, F. (2000). *Automatic Configuration of Component-Based Distributed Systems*. PhD Thesis, University of Illinois at Urbana-Champaign, Urbana, IL.
- Kramer, J., & Magee, J. (1990). The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11). doi:10.1109/32.60317

Maes, P. (1987). Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87, of ACM SIGPLAN Notices* (Vol. 22, pp. 147-155). New York: ACM Press.

Mathy, L., Canonico, R., & Hutchison, D. (2001) An Overlay Tree Building Control Protocol. In *Proceedings of Networked Group Communication*, (LNCS Vol. 2233, pp. 76-87). Berlin: Springer.

Oreizy, P., Medvidovic, N., & Taylor, R. N. (1998). Architecture-based runtime software evolution. In *Proceedings of the 20th international Conference on Software Engineering*.

Rowstron, A., & Druschel, P. (2001). Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proceedings of the International Middleware Conference Middleware*, (LNCS, pp. 329-350). Berlin: Springer Verlag.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., & Balakrishnan, H. (2001). Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, (pp. 149-160).

Szyperski, C. (1998). *Component Software, Beyond Object-Oriented Programming*. New York: ACM Press/Addison-Wesley.

van Renesse, R., Minsky, Y., & Hayden, M. (1998). A Gossip-Based Failure Detection Service. In *Proceedings of Middleware '98, the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, (pp. 55-70).

## KEY TERMS AND DEFINITIONS

**Computational Reflection:** The capability of a system to introspect its own structure and behaviour via a meta-representation, and make changes to this representation that are reflected in the running system.

**Dynamic Adaptation:** The addition, removal or replacement of software elements (e.g. components) at runtime i.e. while the system is performing its core operations.

**Middleware:** System software that typically resides between the application and the operating system that provides a distributed programming paradigm that supports the developer overcome the challenges of distributed computing.

**Overlay Network:** A logical networking infrastructure (topology and routing management) layered over the existing physical network infrastructure.

**Pervasive Grid:** A Grid infrastructure that embraces resources embedded in the environment e.g. in the form of networked sensors and mobile devices.

**Reflective Middleware:** A configurable and reconfigurable middleware platform that uses reflection as a principled mechanism to dynamically adapt middleware behaviour to changing environmental context.

**Software Component:** A third party deployable software module that has private data and a set of provided and required interfaces that explicitly describe the component's behaviour.

## APPENDIX: COMPONENT FRAMEWORKS

We now provide more detailed information about how dynamic adaptation is carried out by software frameworks, and aim to identify the challenges involved for the future development of adaptive Grid software.

Component frameworks manage the adaptation of components contained wholly within themselves. The elements for adapting a component framework can be seen in Figure 8. Here, there are four key elements important to the adaptation process.

- *The architecture meta-interface provides the adaptation operations to the party performing the adaptation. These allow the framework to be inspected and dynamically reconfigured. Hence an adaptation is performed through a transactional sequence of method calls on this interface.*
- *Validation of reconfigurations. Providing open access to a system (through the prior meta-interface) and allowing run-time changes increases the risk of third party attack. To guard against this, each software framework exports a ‘health check’ mechanism through the plug-in interface named IAccept. A component encapsulating knowledge about valid dynamic reconfigurations for this framework is then plugged into the framework here. When a reconfiguration is performed it is validated against this health check; any invalid reconfigurations are blocked and the framework is rolled back to the previous safe state.*
- *A quiescence mechanism places the framework in a state ready to be adapted. For this purpose, each framework provides a readers/writers lock. Every normal interface operation accesses the lock as a reader. Any call through the architecture meta-interface to change the configuration accesses the lock as a writer (a writer can access the lock when there are no readers). Hence, this ensures that there is no executing thread within the framework for the duration of the adaptation.*
- *Configuration Management. The configurator pattern (Kon, 2000) illustrated to the left of figure 8 underpins adaptations. A configurator acts as the unit of autonomy for making decisions about when and how to change the framework (based upon a set of Event-Condition-Action rules stored in a set of policies). When an event is detected (typically a context change notified by the context engine), operations on the architecture meta-interface are called to make the change.*

Figure 8. Component frameworks

