Experiences with Open Overlays: A Middleware Approach to Network Heterogeneity

Paul Grace, Danny Hughes, Barry Porter, Gordon S. Blair, Geoff Coulson, Francois Taiani
Computing Department, Lancaster University,
Lancaster, UK.
{gracep, danny, barry.porter, gordon, geoff, f.taiani}@comp.lancs.ac.uk

ABSTRACT

In order to provide an increasing number of functionalities and benefit from sophisticated and application-tailored services from the network, distributed applications are led to integrate an ever-widening range of networking technologies. As these applications become more complex, this requirement for 'network heterogeneity' is becoming a crucial issue in their development. Although progress has been made in the networking community in addressing such needs through the development of network overlays, we claim in this paper that the middleware community has been slow to integrate these advances into middleware architectures, and, hence, to provide the foundational bedrock for heterogeneous distributed applications. In response, we propose our 'open overlays' framework. This framework, which is part of a wider middleware architecture, accommodates 'overlay plug-ins', allows physical nodes to support multiple overlays, supports the stacking of overlays to create composite protocols, and adopts a declarative approach to configurable deployment and dynamic reconfigurability. The framework has been in development for a number of years and supports an extensive range of overlay plug-ins including popular protocols such as Chord and Pastry. We report on our experiences with the open overlays framework, evaluate it in detail, and illustrate its application in a detailed case study of network heterogeneity.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *Distributed applications*

General Terms

Algorithms, Measurement, Design, Reliability, Experimentation

Keywords

WSN, middleware, overlay network, framework

1. INTRODUCTION

Modern distributed systems can be characterised by increasing levels of *heterogeneity*. This subsumes both the characteristics of the distributed applications and services in question, *and* the environments in which they operate. For example, there are increasing demands for applications that are adaptive, autonomic, dependable, secure, scalable etc., and also demands for such applications to operate in increasingly-varied environments such as the fixed internet, mobile and pervasive environments, embedded systems, etc.

In this paper we address a key aspect of heterogeneity that has perhaps received less attention than it deserves in the middleware community: *network heterogeneity*. As well as needing to run effectively over an ever-increasing range of networking technologies (e.g. large-scale fixed networks, mobile ad-hoc networks, resource impoverished sensor networks, satellite links, etc), distributed applications are increasingly demanding sophisticated and application-tailored services from the underlying network (e.g. multimedia content distribution, reliable multicast, etc.). Furthermore, going beyond this 'classic' view of heterogeneity, we can discern a growing trend towards 'extreme' network heterogeneity involving the combining of already heterogeneous elements. For example [13] discusses scenarios in which sensor networks are tightly integrated with cluster-based and internet-based grids. This trend is also evident in the current interest in systems of systems [43] and the pervasive grid [27].

Such factors have driven the networking community to develop the concept of *network overlays* as an approach to the virtualisation of the underlying network resource(s). Network overlays make it possible to provide a range of different networking abstractions including peer-to-peer groups, distributed hash tables, application-level multicast, etc. In our view, however, this work has not yet been sufficiently embraced and integrated by middleware designers (Several overlay frameworks have been developed (e.g. [32, 8, 2, 38, 33]) but these suffer from significant limitations as discussed in Section 5). We therefore propose the concept of *open overlays* and suggest that it be adopted as a central element of contemporary middleware platforms. In our conception, open overlays offer a configurable and reconfigurable framework that is well integrated into a broader middleware architecture, and supports (flexible) *virtualization* of the network resource, the *co-existence* of multiple (physical or) virtual networking abstractions, and potentially support the *layering* of virtual network abstractions to achieve desired network services through composition.

In this paper we present a detailed *evaluation* of the open overlays approach. This builds on extensive experience of using the approach in the construction and composition of a variety of (often complex) overlays and overlay-based distributed applications. The rest of the paper is structured as follows. Section 2 provides an overview of our open overlays framework, focusing on its associated architectural patterns and its support for

EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.
Copyright 2008 ACM 978-1-60558-013-5/08/04...$5.00.

configuration and reconfiguration. Following this, Section 3 presents an in-depth case-study of network heterogeneity that demonstrates the application of the approach; and Section 4 offers an in-depth discussion of the benefits of the framework in particular and the open overlays concept in general. Finally, Section 5 discusses related work, and Section 6 offers our overall conclusions and plans for further research.

2. THE OPEN OVERLAYS FRAMEWORK

2.1 Context

There are essentially *three* responses to the network heterogeneity that we noted above. The first is to progressively add features to existing middleware platforms to cope with the increased levels of heterogeneity (e.g. extensions to deal with mobile computing). It is now well recognized however that this leads to bloat and is not a viable long-term solution. The second approach is to create specialised per-application-domain middleware platforms (e.g. middleware for sensor networks). This approach has yielded some success but suffers from significant limitations—particularly in terms of achieving interoperability and accommodating the kinds of 'extreme' heterogeneity (e.g. systems that integrate sensor networks *and* clusters) referred to above. The third approach, which we favour, is to offer a *configurable framework* that can be tailored to the needs of a given application and operational domain (or domains) while avoiding the shortcomings of the two previous approaches. Configurable frameworks also have the benefit that they can potentially support run-time *re*configuration, and thus address another emerging trend in modern distributed systems: *dynamicity*, and the consequent need for *adaptivity*.

In general terms, our research over the past few years has been targeting the development of such frameworks through a number of projects including Open ORB [4], ReMMoC [22], NetKit [11], Gridkit [23], and through our contributions to the RUNES middleware [10]. The approach is well documented and builds on the complementary nature of lightweight software component technology (together with component frameworks) in tandem with reflection. Components and component frameworks provide the building blocks and associated principled software engineering methodology for the construction of middleware, and reflection provides the means to inspect and adapt this underlying (explicit) structure, and thus additionally render it reconfigurable at runtime to address the need for adaptivity. OpenCOM [12] lies at the heart of this architectural approach, offering the necessary underlying lightweight and reflective component model.

We have employed this approach in the design of the open overlays framework that is the subject of this paper. The framework is integrated as part of the wider Gridkit middleware architecture [23], which also addresses heterogeneity in other dimensions (e.g. in supporting multiple interaction types [24], and in dealing with heterogeneous service discovery protocols [9]). Aspects of the open overlays framework have previously been presented in the literature [23, 24, 25]; but in the following sub-sections we provide a consolidated overview and update to provide context for the substantial evaluation material in Sections 3 and 4 which forms the main contribution of this paper.

2.2 Basic architecture of the framework

The open overlays framework. The open overlays framework (as visualised in Figure 1) 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:

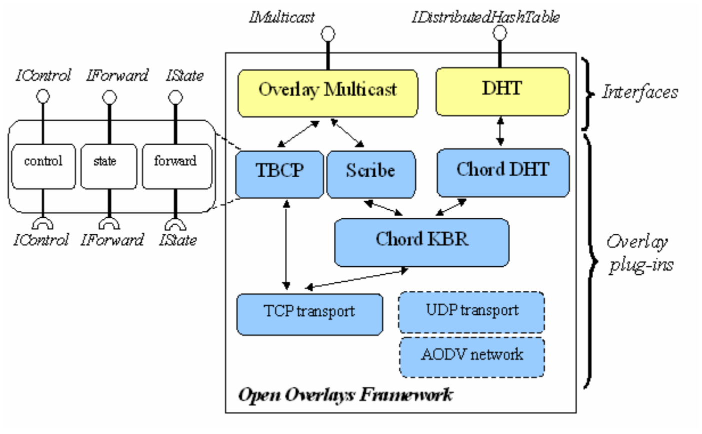


Figure 1. An example configuration of the open overlays framework

i) *Overlay plug-ins*. These are per-node implementations of network overlays. For example, Figure 1 shows four overlay plug-ins: TBCP [35], Scribe [7], and plug-ins for a Chord Distributed Hash Table (DHT) and a Chord Key-Based Routing (KBR) overlay [44]. Multiple overlays can operate simultaneously in the framework either in mutual isolation (cf. TBCP and Scribe in Figure 1) or in a stacking relationship (e.g. Scribe and Chord DHT are both stacked atop Chord KBR). The overlay plug-in abstraction can be applied uniformly throughout the communication stack. For example, transport protocols like TCP or UDP are represented as overlay plug-ins, and an AODV overlay plug-in may be provided in the network layer in a MANET environment. Note, we term plug-ins implementing transport behaviour (i.e. no routing) as null overlays. Hence, the abstraction can even be applied at the level of the physical network as demonstrated in Section 3.

ii) *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. For example, following [15], we provide an interface plug-in for DHT overlays and another for 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 (i.e. transparently replacing one overlay with another), and encourages a principled approach to the development of 'families' of overlays plug-ins, each of which shares a common API.

A pattern for overlay plug-ins. Overlay plug-ins are themselves 'mini' component frameworks (in OpenCOM, component frameworks are inherently components), each of which, as shown in the left part of Figure 1, is composed of three distinct elements (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. Note also that the overlay pattern can form a basis for further decomposition—i.e. each of the three elements can itself be a component framework. We consider such an overlay in Section 4.

2.3 Local configuration and reconfiguration

Local configuration Each per-node instance of the open overlays framework is dynamically configured at deploy-time. Possible configurations are first set out in terms of a set of pre-installed *profiles*, each of which specifies an available palette of overlay and interface plug-ins and a set of basic constraints that specify configurations that are recognised by the profile. As examples, we have defined profiles for multicast environments and for wireless sensor networks (see Section 4, table 6).

To support configuration, the framework employs both static and dynamic meta-data as follows:

i) *static* meta-data is attached to the set of overlay plug-ins currently available in the profile; this specifies a set of *configuration rules* (see below), constrains which other overlay plug-ins may be stacked below the plug-in, and may also constrain which interface plug-in the overlay requires;

ii) *dynamic* meta-data is provided by a per-node *context engine* [Capra,03]; this meta-data varies dynamically according to the current state of the host node in terms of relevant characteristics such as battery life, network connectivity etc.

The static configuration rules contained within each profile are declarative XML-based expressions that specify the configuration possibilities supported by the profile. As an example, the following configuration rule (expressed in pseudo-code rather than XML for the sake of clarity) states 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—i.e. to match one of the configurations shown in Figure 1:

```
if (multicast && fixed_infrastructure &&
   !IP_multicast)
    configure overlay_multicast interface with TBCP
```

Once the execution of such a rule has resulted in the instantiation of a 'top-level' overlay plug-in (i.e. TBCP in our example), the configuration process continues in a delegated manner which we refer to as *top-down recursive instantiation*. This involves each overlay plug-in evaluating its own configuration rules, and on that basis selecting, instantiating (or discovering) and configuring a further overlay at the next level down. This process continues until an overlay plug-in is encountered which has no rules that trigger any further instantiation.

Local reconfiguration Having established a configuration as discussed above, it is possible to dynamically reconfigure a node's overlay configuration using the 'standard' OpenCOM reflective capabilities [12]. For example, one can inspect the current composition of components in the framework, replace or add components, or add interceptors. However, in line with the usual semantics of OpenCOM component frameworks, all such actions are subject to 'veto' if they would violate the meta-data constraints associated with the profile or the current state of the framework instance. For example, a constraint of the overlay framework is that there must be a null overlay plug-in (e.g. transport or physical network component) at the bottom of the overlay framework configuration.

2.4 Distributed deployment & reconfiguration

While local configuration and reconfiguration rely on static and dynamic meta-data available on each node, *Distributed* configuration (i.e. overlay deployment) and reconfiguration both rely on generic support provided by OpenCOM's *distributed component framework* (DCF) facility [25].

DCFs are coordinated sets of local component framework instances that are spread across a set of co-ordinated nodes. For example, a DCF-enabled extension of the TBCP overlay plug-in from Figure 1 would contain an instance of the TBCP component framework for every node participating in the overlay. DCFs support dynamic reconfiguration both at a coarse-grained (e.g. changing the top-level overlay in use), and a fine-grained level (e.g. changing the overlay plug-ins underlying the top-level one, or changing one of the elements within an individual overlay plug-in).

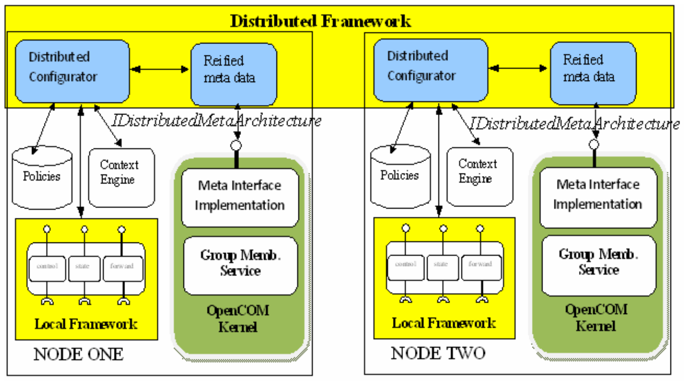


Figure 2. The per-node elements of a Distributed Component Framework

The DCF facility is supported by the per-node architecture illustrated in Figure 2. Briefly, both deployment and reconfiguration are driven by *configurators* which select and apply reconfiguration 'policies'—i.e. scripts to be executed on each DCF node to enact a specified deployment or reconfiguration action. The selection of these policies from a policy repository is performed similarly to the local configurations/reconfigurations discussed in section 2.2 (i.e. based on meta-data and configuration rules).

DCFs themselves can be very flexibly configured according to application needs. For example, depending on the numbers of participating nodes, each DCF may employ a single master configurator or per-node distributed configurators. Similarly, they may employ either a single or multiple context engine and policy database. In addition, the strategies used to achieve consensus in the case of distributed configurators, or to achieve quiescence before applying a policy script, can all be flexibly configured. We also support configurable strategies to post-validate policy enactments, ranging from simple but scalable strategies based on exceptions to a fully reliable (but not very scalable) transaction protocol. Each DCF also maintains a meta-interface (see *IDistributedMetaArchitecture* in figure 2) that enables the atomic insertion (deletion) of components into (from) the local component framework instances on all the participating nodes. The meta-interface also reifies information about the DCF in terms of its participating nodes and their current component configurations. The communication underlying the meta-interface is implemented in terms of a lightweight group membership service [21].

For safe dynamic reconfiguration it is important to ensure that updates do not impact the integrity of the system. Hence, the distributed framework must be made safe to adapt, i.e. placing it in a quiescent state. We have so far developed a single, centralised implementation for deriving a safe state in the distributed framework (this is used for the evaluation results in section 3). A request to reconfigure the distributed framework from a central node generates a request message asking each local framework instance to be placed in a quiescent state; this message is propagated via gossiping through the meta-group service. Once a local framework is in a quiescent state it returns a notification to the configurator node. Upon the condition that all members are in a quiescent state the reconfiguration can take place. The disadvantage of the centralised approach is that it may be too resource intensive, and may not scale suitably for large numbers of nodes. Additionally, it may not be necessary to place all nodes in a safe-state at the same time, or have a single node managing the transition to a safe state. Hence, our framework also supports selectable approaches to safe-state management that can be tailored to the particular style of reconfiguration to be performed and the environment that the framework is deployed.

In sum, in the context of the open overlays framework the DCF is used to make coordinated changes across all member nodes of an overlay. For example, in a spanning tree overlay plug-in we can in one action change the topology of the overlay from a 'fewest hop' to a 'shortest path' configuration by reconfiguring the control element of the plug-in on each node (see Section 3). Similarly, we might change the routing strategy of a multicast overlay to anycast by globally reconfiguring the forwarding elements.

3. CASE-STUDY BASED EVALUATION

3.1 Background

We now discuss the application of the open overlays framework in an implemented real-world scenario: wireless sensor network-based real-time flood forecasting in a river valley in the north west of England. This work has previously been published from an application perspective [28]; this paper, in contrast, takes a quantitative perspective and focuses especially on the *dynamic reconfigurability* capabilities of the open overlays framework in the scenario.

In terms of necessary background, we monitor water depth and flow rate in the river by deploying a number of specialised sensor nodes along the banks of the river. About 15 nodes are currently deployed. The sensor data is collected in real-time and routed using a spanning tree topology to one or more designated 'root' nodes. From there the data is forwarded via GPRS to a prediction model that runs on a remote computational cluster.

Each sensor node (known as 'GridStix') comprises a 400MHz XScale CPU, 64MB of RAM, 16MB of flash memory, and Bluetooth and WiFi networks (the root nodes are also equipped with GPRS). 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. Unlike traditional sensor network deployments, wherein sensors are merely responsible for relaying sensor data to off-site processing facilities, this deployment makes significant use of local processing, which is used to support computationally complex sensors and to support the local prediction of future environmental conditions. This functionality 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, 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.

To support these heterogeneous application requirements, we have implemented *a tailored Flood-WSN profile* on top of our Overlays Framework, which we deployed on each node in the network. In the remainder of this section we describe in more details how we mapped our application requirements unto this domain-specific profile. We also present lines-of-code (LoC) and memory footprint measurements to convey an idea of the size and complexity incurred by this implementation of this profile.

In a second part (Sections 3.3 and 3.4), we then discuss the overall performance of the resulting system, in terms of *latency*, *resilience* and *power consumption*. In particular we look at the impact of the reconfigurations made possible by the framework. More specifically, the figures we present show that the use of the framework has no detrimental impact on the overall performance of the flood prediction network, and that the reconfiguration mechanisms embedded in the framework cause acceptable overheads, two crucial preconditions for the deployment of our technology in real applications.

3.2 The Flood-WSN profile

Our application supports reconfiguration along two dimensions, which both lend themselves to the structures offered by our Overlays Framework:

i) **At the physical network level** each node can use either *Bluetooth* or *WiFi* (802.11b). Both technologies have extremely different throughput, energy, and range properties as summarised in Table 1 (These power draw figures are based on Ericsson ROK-104-001 BT modules, and Marvell 88W8385 WiFi modules. The given range figures were measured using strategically deployed directional antennas). WiFi provides the highest throughput and longest range, but at the cost of energy consumption almost an order of magnitude higher than Bluetooth. Typically Bluetooth would be used in quiescent conditions, and WiFi in imminent flooding situations.

ii) **At the data routing level** data may be routed from the sensor nodes to the root node along two different types of spanning tree: either using a 'shortest path' (SP), or a 'fewest hop' (FH) strategy. *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;

Table 1. Relevant characteristics of Bluetooth and WiFi

	Throughput	Power Draw	Range
Bluetooth	786Kbps	0.4W typical	up to 200M
WiFi	11Mbps	2.9W typical	up to 1.2KM

These two levels of optional configuration are reflected in our Flood-WSN profile by four options (WiFi, Bluetooth, SP and FP spanning trees). As FH and SP overlays differ only in terms of their forwarding components, an FH overlay may be implemented simply by creating a new forwarding component and re-using the state and control components of the SP tree.

The storage memory footprint (on disk) of the resulting code is shown on Table 2. The Flood-WSN profile consumes just 28KB of storage memory, and an average of 105KB of dynamic memory during execution inclusive of platform specific overheads such as the Java virtual machine running on the GridStix. In order to save dynamic memory, overlays are instantiated on demand, rather than being maintained concurrently.

Table 2. Footprint of the WSN Profile in deployment.

	Storage Memory
OpenCOM	52.4KB
Overlays Framework	23.8KB
Flood-WSN Profile	28.0KB
Total	104.2KB

The WiFi/Blueetooth capabilities were extremely easy to implement as they directly rely on OS-level capabilities. Much more interesting for the assessment of the Overlays Framework is the implementation of the two types of Spanning Trees topologies, whose size and footprints are described in Table 3. The spanning tree plug-ins necessitated the creation of four classes on top of the underlying framework, one for each element of the overlays pattern that we presented in Section 2.2. Two classes served both spanning trees (the control and state components), with two differentiated forwarding components were implemented, one creating an SP tree, and one an FH tree. Importantly, as Table 3 shows, this re-use of components between FH and SP trees allows an additional tree overlay to be implemented by replacing a single component at a storage cost of only 6.5KB.

Table 3. Breakdown of Overlay Memory Footprint

	Shortest Path (SP)		Fewest Hops (FH)	
	Size	***LoC***	***Size***	***LoC***
Control	7.3KB	124	*re-used*	*re-used*
State	2.5KB	24	*re-used*	*re-used*
Forward	6.3KB	346	6.5KB	352

3.3 Overall Performance

We now discuss the overall performance of the resulting application, in terms of *latency*, *resilience* and *power consumption*. More precisely, we start with a quantitative evaluation of the relative costs and benefits of the various options identified above as a basis for determining the conditions under which the system might best be reconfigured. We first discuss how the use of Bluetooth or WiFi influences the characteristics of the network, and then move on to compare the two different spanning tree configurations (Fewest Hops and Shortest Path). The criteria employed include both generic metrics and application-specific concerns (see below). The generic metrics are as follows:

i) *Latency*: We quantify this in terms of the average latency with which messages can be relayed from each sensor node to the root node (and thence to the back-end flood prediction models).

ii) *Resilience*: This is a function of the extent to which the failure of a given node reduces the overall connectedness of the network. We quantify it as the number of viable routes between each node and the root.

iii) *Power Consumption*: Although the GridStix are equipped with solar panels, power consumption is still an extremely important factor given that flooding occurs in conditions of low light intensity! We quantify this as the per-hop power consumed during the transmission of a 1KB sensor reading from each node to the root.

In all cases we measure and plot each of these metrics for each node in the network. The figures were obtained by empirical measurements on a lab version of the deployed system with a topology as shown in Figure 3.

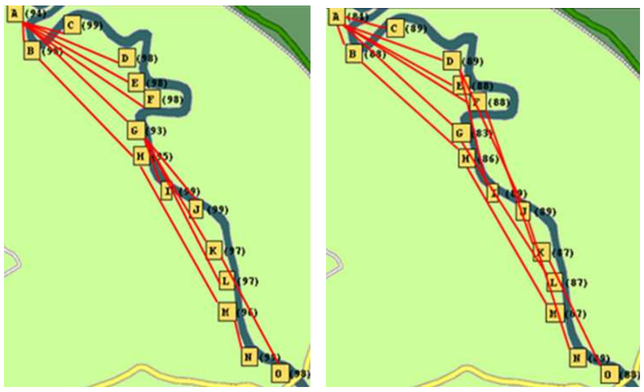


Figure 3. FH (left) and SP (right) Spanning Trees

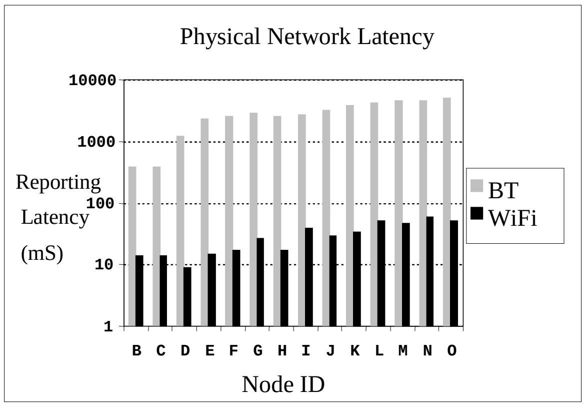


Figure 4a. Physical Network Latency

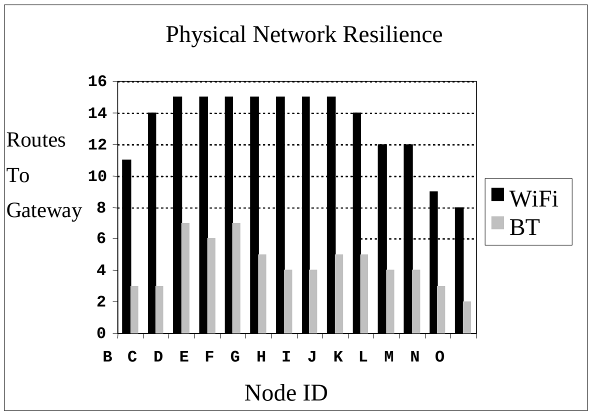


Figure 4b. Physical Network Resilience

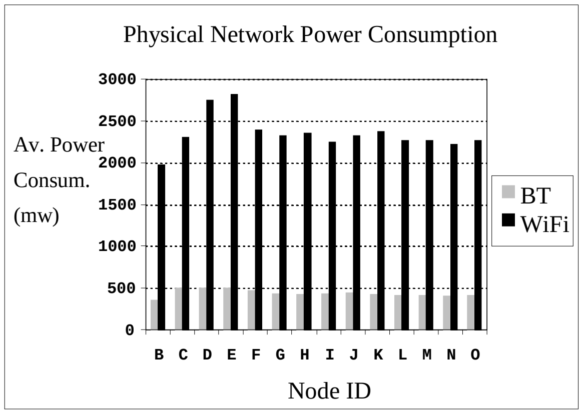


Figure 4c. Physical Network Power Consumption

The Bluetooth and WiFi configurations were evaluated against each other using a common configuration at the Spanning Tree level in both cases (we chose an SP configuration that is designed to minimise power consumption). In Figure 4a, the 'latency' graph shows that WiFi incurs significantly less latency than Bluetooth (over nodes B-O—i.e. 14 non-root nodes): average reporting latency for the latter was 2,912ms, compared to just 38ms with WiFi. The 'resilience' graph (in Figure 4b) again shows Wifi performing significantly better than Bluetooth: the average number of routes from each node is 13.2 for Wifi compared to just 4.4 for Bluetooth. Finally, the 'power consumption' graph (in Figure 4c) shows that Bluetooth consumes significantly less power than the lowest power WiFi configuration. The average per-hop power consumption was 0.44 Watts for Bluetooth and 2.35 Watts for WiFi.

In summary, and as expected, WiFi offers lower latency and higher resilience than the Bluetooth configuration, but consumes significantly more power. It is also interesting to note that for each of the three properties evaluated there are significant variations across the nodes. This implies that a decision as to the optimal time at which a reconfiguration operation should be initiated ought ideally to be informed by data from multiple nodes in the tree. Last but not least, the energy figures are in lines with those measured in Table 1 - the power consumed by local computation during reconfiguration is negligible compared to the power consumed due to network use.

Spanning Tree configurations The SP and FH overlay network configurations were evaluated against each other (using the WiFi network configuration in both cases); the results are seen in Figure 5.

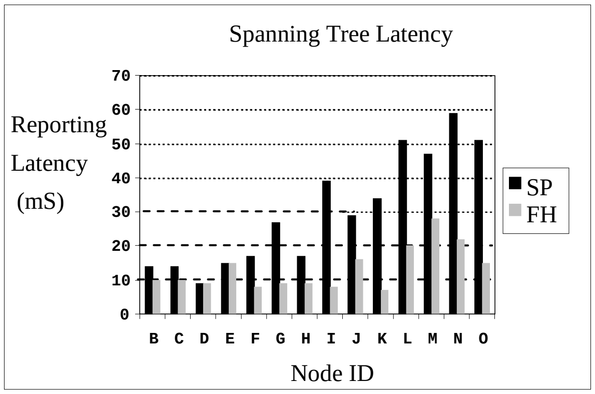


Figure 5a. Spanning Tree Latency

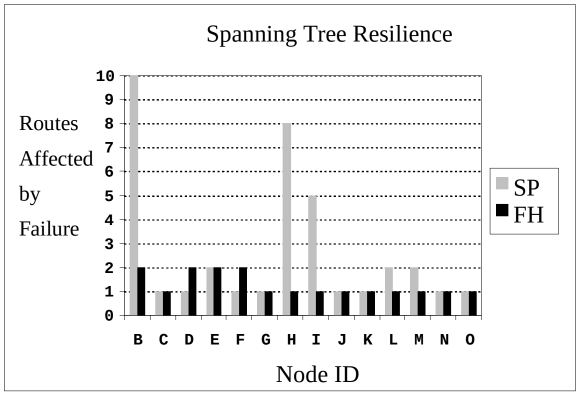


Figure 5b. Spanning Tree Resilience

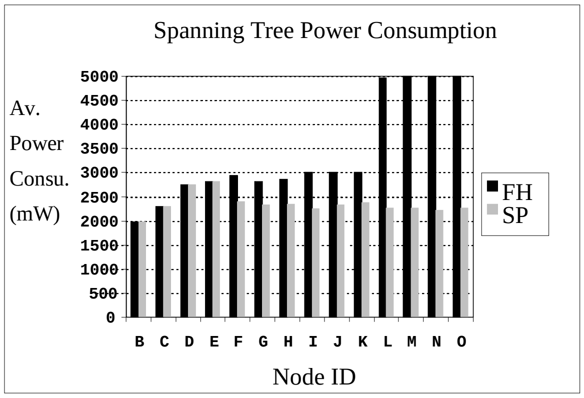
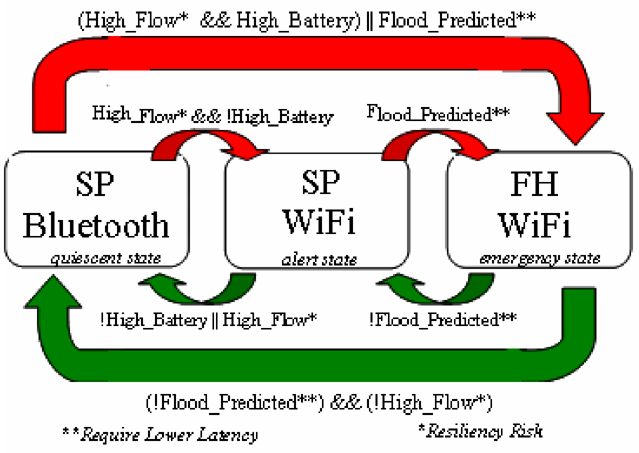


Figure 5c. Spanning Tree Power Consumption

In Figure 5a, the 'latency' graph shows FH performing significantly better than SP: the average reporting latency with FH was 11 milliseconds compared to 28 milliseconds for SP. As expected, reporting latency in both cases tends to increase with separation from the gateway node (the nodes to the right of the bar chart happen to be those that are physically located furthest from the root). On the 'resilience' graph (in Figure 5b) FH again performs significantly better than SP: the average number of nodes affected by node failure in FH was 1.29, as compared to 2.64 for SP. Finally, the 'power consumption' graph (in Figure 5c) shows that FH consumes significantly more power than SP: the average per-hop power consumption was 3.39 Watts for FH and 2.35 Watts for SP.

In summary, FH is significantly better than SP in terms of latency and resilience, but consumes significantly more power. Again there are significant variations from node to node. These differences between the FH and SP topologies, and between individual nodes tends to show that the main drivers for these measurements lay outside of the Overlay Framework and the Flood-WSN profile, whose impact is probably negligible compared to other influencing factors, such as the lengths of routes, and the characteristics of the wireless technologies in use.

Triggering Reconfiguration: Reconfiguration is supported in our sensor network though the Distributed Component Framework facility included in the Overlays Framework (see Section 2.4). The reconfiguration opportunities arising from the above analysis, and the associated '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). We also show a representative pseudo-code configuration rule relating to one of the transitions (the top one).



```
If (High_Flow && High_Battery)
      || Flood_Predicted
   replace Bluetooh with Wifi
   replace ST.sp with ST.fh
```

Figure 6. Reconfiguration states and triggers (simplified)

As can be seen, the triggers/rules are partly based on the factors of latency, resilience and power consumption discussed above; but they also include two additional application-specific triggers. The first of these, *High_Flow*, is based on 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. In the other, *Flood_Predicted*, the trigger is provided by so-called point prediction models [3] which provide localised predictions of water depth based on the collated readings of depth sensors in the immediate locality. Interestingly, the computations underlying *High_Flow* and *Flood_Predicted* run in a distributed manner *on the GridStix* *nodes themselves*; and the open overlays framework is used to instantiate additional overlays to handle the coordination involved in these distributed computation (cf. the principle of supporting multiple co-existing overlays).

3.4 Evaluating the cost of reconfiguration

While dynamic DCF-based reconfiguration clearly enables system utility to be optimised for varying environmental conditions, there is a cost associated with each reconfiguration operation in terms of the time taken to perform the reconfiguration, and the power consumed by the additional CPU and network activity. These are significant costs: time taken reconfiguring is time during which the network is out of commission (involving lost sensor readings); and consuming additional power clearly increases the risk of losing nodes due to power depletion. We therefore carried out experiments to evaluate the cost of reconfiguration in our case study scenario. We focused on reconfiguration at the Spanning Tree level as this avoids network-specific overheads that would inevitably impact measurements involving switching between the two physical networks.

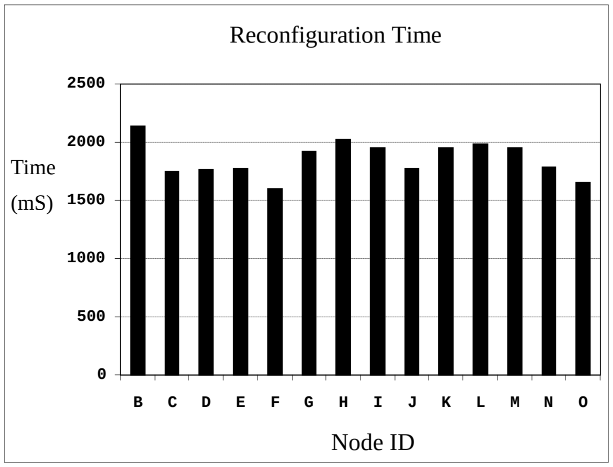


Figure 7a. Reconfiguration times

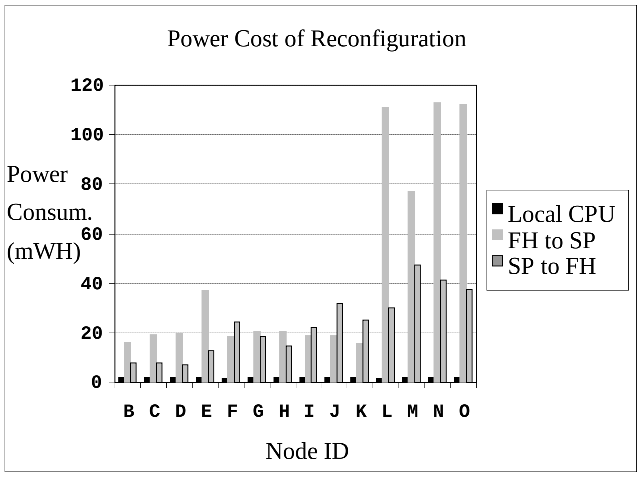


Figure 7b. Reconfiguration costs in terms of power

Figure 7a shows the per-node time required to reconfigure between FH and SP spanning trees for the same topology that was used in Section 3.2. The average time required was 1878 ms. To put this in perspective, at typical sensing rates used in the case study this implies an average loss of less than one message (0.36 messages).

Figure 7b shows the per-node power costs associated with reconfiguration from FH to SP and vice versa (there is a difference in cost between the two directions because the DCF was configured to employ the current overlay to support reconfiguration operations). When reconfiguring from FH to SP, network power costs an average of 44.2 milliwatt hours of battery life per node. When switching from SP to FH, network power costs average 23.5 milliwatt hours. But to put this in perspective, the maximum power consumed during overlay reconfiguration for any node (46 milliwatt hours) is equivalent to less than 0.05% of the battery capacity of a GridStix, which in combination with the infrequent nature of reconfiguration is effectively negligible.

Overall, it can be seen that the power and time overheads of reconfiguration are relatively small, particularly compared to the potential benefits of optimising the system to current conditions.

For comparison, and in order to apportion the share of the Framework's mechanisms in reconfiguration costs, average overheads for local reconfiguration in the Framework are provided in Table 4. This shows that the frameworks reconfiguration overhead is quite reasonable, averaging 198 ms, and roughly only represents one tenth of the reconfiguration time observed on each node, the rest being due to the latency of message passing that is required to support distributed coordination, and thus outside of the Open Overlays structure.

Table 4. Time overhead of Reconfiguration.

	Overhead (ms)
Component Creation	118
Component Binding	69
Component Connection	11
Total	198

3.5 Conclusion

This case study shows that different level of network heterogeneity (here both in wireless technologies, and in overlay topologies) can easily be captured into the structures of our Overlays-Framework. The decomposition of overlay plug-ins into three standard components also encourages reuses, and allows developers to support a wide scope of alternative configurations at a relatively low cost in terms of memory footprint and implementation effort.

Finally, the performance figures we presented show that the use of the framework has no obvious detrimental impact of the overall performance of the flood prediction network, and that the reconfiguration mechanisms embedded in the framework cause acceptable overheads, two crucial preconditions for the deployment of our technology in real applications.

4. LESSONS LEARNT AND DISCUSSION

In this section, we expand on the conclusion of the case-study evaluation and we present the lessons we have learnt on the benefits of our Open Overlays framework as we applied it to a number of different domains. We discuss qualitatively the advantages of the framework in terms of software development, and present some more extensive performance figures in terms of memory footprint and configuration times for some of the many plug-ins we have already implemented.

We finish this section with a general discussion of the value we see in the fundamental notion of overlays as an architectural principle for heterogeneous middleware.

4.1 Benefits of the open overlays framework

We evaluate the effectiveness of the open overlays framework against the following four criteria:

i) *Generality*: To what extent can the framework be generally applied in terms of different network services deployed in different network environments; and how general is the overlay pattern in developing overlays?

ii) *Ease of Use*: How easy is it for a developer to use the framework, and extend it with new functionality?

iii) *Configurability*: To what extent can the framework be configured to meet specific requirements and environments (an in-depth evaluation of *reconfigurability* is provided in Section 3)?

iv) *Resource Overhead*: Is the overhead incurred to support generality, configurability and reconfigurability acceptable?

Generality As an indicator of generality, we have developed a substantial set of overlay plug-ins of which Table 5 lists eight. From this list, the generality in terms of the network services provided is clear: we cover 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. Table 5 also shows the configurability options offered by each overlay plug-in (in brackets, following the descriptions), and illustrates that the framework can be generally applied in different network environments (thus addressing network heterogeneity): e.g. we can use the Spanning Tree overlay in a wireless sensor network (see Section 3); and we can choose different multicast protocols for different networks: e.g. TBCP for wide area networks, or Scamp for wireless networks.

These various implementations also provide a strong and comprehensive evaluation of the overlay pattern: all eight were straightforwardly implemented in terms of the three defined elements (i.e. control, state and forward), providing clear evidence that the pattern applies generally to different overlay types. Moreover, each of the implementations exhibits a clear and natural separation of concerns in contrast to many monolithic implementations.

As shown in Figure 8, one of the eight overlays, i.e. Pastry KBR, was further decomposed to investigate finer-grained configurability and reconfigurability (for example, the control element is composed of sub-components corresponding to distinct Pastry algorithms, i.e. for joining, leaving, maintenance and repair). This decomposition demonstrates that the overlay pattern can itself be extended to meet the complexities of individual overlays and yet still be supported by the framework.

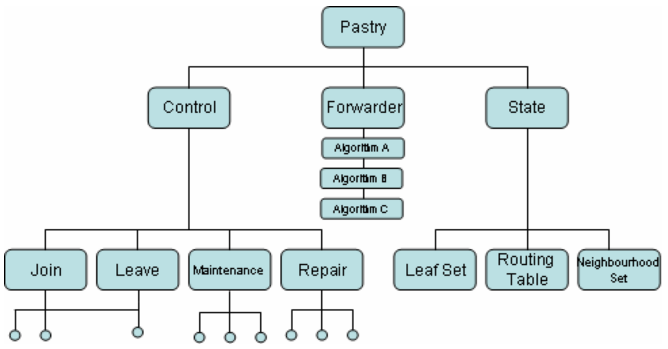


Figure 8. Extending the overlay pattern in the Pastry KBR plug-in

Table 5. Descriptions of some implemented overlay plug-ins

Overlay Name	**Description and configurability options**
Chord KBR	A KBR overlay based on Chord [44] (options: standard or 'dependable' control element; 2 choices of supporting overlay)
DHT	Data storage overlay (options: standard or 'dependable' control element; used atop any KBR overlay)
Pastry KBR	A KBR overlay based on Pastry [41] (options: supports alternate overlay maintenance algorithms)
Failure Monitor	Monitoring overlay based on [45]; detects and disseminates node failure info (options: used atop any gossip overlay)
SCAMP	Scaleable Group Membership overlay with gossip-based forwarding [21] (options: 2 choices of supporting overlay)
Scribe	Multicast based on [7] (options: used atop any KBR overlay)
Spanning Tree	Tree overlay for fan-in routing (options: shortest path or fewest hop tree configurations; used atop either Wifi or Bluetooth 'overlays')
TBCP	Wide area multicast overlay [35] (options: standard or 'dependable' control element; 2 choices of supporting overlay)

Ease of use The framework has been used by over 15 programmers, from a range of institutions, with different levels of programming experience, in a number of system development projects (e.g., projects developing middleware for sensor networks, resource discovery, and publish-subscribe). Some of these programmers contributed as 'plug-in developers', some as 'framework configurers and users', and some as both. From observation and discussion were able to draw the following conclusions:

i) Plug-in developers generally understood and followed the approach implied by the overlay pattern, and to this extent their solutions are easily deployable by third party application developers. A caveat is that in some cases control, forward and state were not completely separated into distinct components. This is an area where further software engineering support might benefit both the plug-in developer and the framework configurer/ user.

ii) A typical overlay plug-in is developed in a time frame of 2 to 8 weeks depending on the complexity of the overlay.

iii) Framework users found it relatively easy to apply the existing profiles of the framework; but in cases where new configuration rules needed to be defined, they expressed the need for clearer documentation of the set of attributes and context values understood by the framework.

Hence, despite the fact that the evidence is primarily anecdotal, and that there are areas of possible improvement, we believe that it is reasonably safe to conclude that third parties can use the framework with relative ease.

Configurability To measure the extent of the configurability of the framework we calculated the numbers 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, a 'multicast' profile for multicast overlays, and a 'full' profile containing all of the foregoing; see Table 6). The numbers, which are summarised in the rightmost column of Table 6, result from an exhaustive enumeration of all the configurations reachable via the 'top-down recursive instantiation' process described in Section 2.3, applied to the set of plug-ins available in each profile (for example, TBCP can be configured with either a standard or a 'dependable' control element, and it can be layered over TCP and UDP transport 'overlays' and thus yields 4 configurations at its level). 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; this does not mean that programmers must write 27,000 rules, rather the approximately 30 rules for the full profile combine to offer many potential configurations. But, more importantly, because of the top-down recursive instantiation process, all of these configurations are meaningful. This is because the architecture of the framework disallows invalid instantiations. This can be compared to other configurable toolkits such as Ensemble [46] or JGroups (www.jgroups.org) which, despite supporting millions of combinations, offer a much smaller number that are actually useful (because these use event-based component bindings that allows components to be connected to any other in any order).

Furthermore, the overlay pattern contributes significantly to the configurability of the framework by supporting 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. This applies equally when the overlay pattern is decomposed. For example, our Pastry example above supports two alternative implementations of the maintenance sub-component: one version, which is based on the original Pastry algorithm, employs frequent leaf set broadcasts over TCP connections; the other employs UDP-based keep-alive messages to monitor the state of its leaf set. The latter algorithm is less robust to network wide failure or malicious attack, but generates far fewer network messages.

Resource overhead To assess the price paid for its generality, ease-of-use and configurability, we quantified the resource overhead incurred by the open overlays framework in three experiments. All of these employed components from Gridkit 1.5/ OpenCOM v1.3.5 (available from http://gridkit.sourceforge.net), 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 6) investigated the *static storage footprint* costs of each profile; i.e. the disk space required to store the framework, components and configuration rules. This measure is important as it illustrates the cost of storing not only a starting configuration but also any reconfigurations that may subsequently be applied. It can be seen from Table 6 that the base framework requires 60K before any plug-ins are added. Note that the configuration rules take a lot of storage (usually at least 2KBytes) because they are coded in XML. A more efficient representation might be better for profiles that are both complex and designed to be applied in resource-scarce environments.

Table 6. Configurability results and overheads for framework profiles

Profile	No. plug-ins	No. config. rules	Disk mem. for config. rules (KB)	Disk mem for plug-ins (KB)	Total No. of configs available
Empty	0	0	0	60	1
WSN	7	6	16	146	4
Multicast	21	19	59	169	89
Full	40	31	87	252	26,999

Table 7. Performance times and dynamic memory costs of typical configurations

Configuration Name	#plug-ins	#Conns	Profile	Config. time (ms)	Dynamic mem. (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
Spanning tree	5	12	Full	193	15,264
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

In the second experiment (see Table 7) , 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 from Table 7 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 (different profiles will have different numbers of configuration rules in memory). More complex configurations, e.g. the layering of Scribe over a Chord KBR, or TBCP and Scamp deployed in parallel, 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 7) 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 (including rule evaluation and component initialisation) 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 costs of (DCF-based) distributed configuration (i.e. overlay deployment) need not be much more costly than this depending on the protocol used (e.g. Scamp [21]).

4.2 Assessing the open overlays concept

To evaluate the open overlays concept, we examine how successful we have been in achieving the desired properties of virtualisation of the network resource, co-existence of overlays, and layering of overlays to compose network services. In terms of *virtualisation*, Section 4.1 has illustrated the range of overlays that can be virtualised by common interfaces, e.g. multicast and DHT network resources. These have then been utilised to build a wide range of higher level middleware services, e.g. publish-subscribe, group middleware [24] and sensor middleware (see Section 3) that are independent of the network service; i.e. the middleware can be deployed in different networked environments without modification. This work has demonstrated that virtualisation is indeed a powerful concept when incorporated within an overall (configurable and reconfigurable) middleware architecture. This is also something that is quite unique in that existing middleware platforms/ paradigms do not yet support network virtualisation.

In terms of *co-existence*, our experience shows that i) overlays can be deployed in parallel, ii) this is indeed a useful service to offer, and iii) the overheads of co-existence are reasonable. This is best illustrated by the sensor network scenario in Section 3, which utilises three separate overlays. One outstanding issue is the *management* of co-existence, in particular in terms of QoS properties. We are currently investigating the potential role of the work by Cooper [8] in addressing this problem. Finally, in terms of *layering* we have shown that relevant overlays can usefully be stacked on top of each other, e.g. the Scribe overlay (or the DHT) can be stacked on top of either Pastry or Chord. Similarly, in the scenario in Section 3 we layer a Spanning Tree overlay on top of either Bluetooth or 802.11b networks. The layering process is guided by top-down recursive instantiation and the use of uniform interfaces, and promotes the reuse of lower-level overlay plug-ins.

5. RELATED WORK

Specialised middleware As mentioned in the introduction, one approach to dealing with heterogeneity is to develop a series of specialist middleware platforms for particular domains of operation. This approach has been most prevalent in the mobile computing domain, with a wide variety of platforms emerging including: context-aware and adaptive technologies [6, 37]; particular interaction paradigms [16, 14]; and more specific techniques to deal with disconnection [29]. Some interesting techniques have also emerged to deal with heterogeneity in service discovery platforms, including the ReMMoC platform from Lancaster [22] and the INDISS work at INRIA [5]. Specialist middleware technologies have also emerged in areas such as distributed multimedia [47, 17] and grid computing [19, 20].

There is currently strong interest in middleware for sensor networks. This is a relatively new development building on the early experiences with operating systems in this area. Middleware approaches for wireless sensor networks seek to provide abstract programming models that offer a more global distributed systems management perspective, often enabling multiple applications to co-exist and share the underlying sensor infrastructure. A good survey of middleware for sensor networks can be found in [26], which includes a taxonomy for sensor middleware featuring database-inspired approaches, tuple-space approaches and event-based approaches as important sub-classes.

It is clear that significant advances have been made in terms of specialist techniques for particular environmental or application domains. Despite these advances, though, the specialist middleware approach has a number of very significant limitations. In particular, these solutions remain narrow in scope and do not help with problems such as interoperability with other domains. In addition, they are all developed independently of each other and there is no support for the re-use of software in other domains, i.e. there is no common architectural framework.

Configurable and reconfigurable middleware There has been considerable interest over the last decade in techniques that support configurability and reconfigurability in middleware. Such techniques typically rely on underlying reflective support including both structural and behavioral reflection. Examples of key reflective middleware platforms include the work at Lancaster mentioned above, the families of platforms developed at the University of Illinois at Urbana Champaign [30, 39], ExORB [40], Arctic Beans [1] and RAPIDWare [42]. This paper follows this general approach and reflection lies at the heart of our proposal for open overlays.

Other researchers are investigating the potential role of aspect-oriented programming in supporting configurable (and in some cases reconfigurable) middleware platforms [48, 31]. This is in many ways complementary to reflective middleware, seeking higher level aspect-oriented constructs to express the weaving of cross-cutting concerns in middleware. Indeed, some implementations in this area build on top of reflective middleware technology [18].

Overlay Frameworks As mentioned in the introduction, the networking community has been carrying out a significant volume of research directed towards the development of network overlays. However, most of this research has been targeted towards the implementation of application specific protocols such as peer-to-peer substrates or multicast solutions. In this section, we focus on the smaller number of initiatives focusing on middleware frameworks to support overlay software.

iOverlays [32] was one of the earliest attempts to define a framework for the support of overlay networks. Essentially, iOverlays is low-level software cross-connect that forwards messages according to a script that embodies the semantics of a particular overlay. ODIN-S [8] also provides a framework for overlay development, with an emphasis on managing resources for overlays that share common nodes, i.e. co-existent overlays. As such this work is strongly complementary to ours. [34] also explores the co-existence of multiple overlay networks across nodes, in particular demonstrating that the maintenance and deployment of one overlay (in this case Pastry) can utilise the behaviour of another (a gossip protocol) to improve its operation. Both these systems illustrate use-cases that can be generally developed using our open overlays framework.

Other solutions target the declarative description of overlay networks and the subsequent automatic generation of code to implement the desired virtual network abstraction (cf. model-driven engineering) [2, 38, 33]. This is an interesting approach to managing the complexity of configurable middleware. The above solutions focus almost entirely on *configuring* overlay networks, i.e. on tool support for the generation of a given overlay style. There is little or no work on the subsequent management of overlays, specifically reconfiguration as context changes. In addition, all of the above are stand-alone toolkits and are not integrated into broader middleware architectures. We acknowledge that currently our framework is not as declarative as these approaches in that low-level overlay code (e.g. in Java or C++) must be created first before being plugged into our framework. However, we do believe the two approaches are complementary and we are investigating model-driven approaches for generating code to insert into the open overlays framework.

6. CONCLUSIONS AND FUTURE WORK

We have presented and evaluated our concept of open overlays and its associated framework—a framework that is designed to comprehensively address the 'network heterogeneity' problem in the context of middleware architecture. In our evaluation of the open overlays concept, we have argued for the usefulness of network virtualisation in a middleware context, the usefulness of supporting multiple overlays per node and the stacking of overlays, and the benefits of structuring overlay plug-ins according to the overlay pattern. In terms of the framework-specific evaluation, we have focused on the framework's generality, its ease of use for both plug-in developers and configurers/ users, the practicability of its configuration and reconfiguration capabilities (employing top-down recursive instantiation, declarative rule-based configuration and reconfiguration, and distributed deployment and reconfiguration), and the fact that it incurs only a modest resource overhead. Furthermore, we have presented a detailed case study of network heterogeneity and at the same time demonstrated the use of our framework in a challenging, WSN-based, application context in supporting multiple overlays, and dynamically reconfiguring them according to current environmental conditions.

In current work, we are integrating an overlay-independent *dependability subsystem* [36] into the framework. This can significantly simplify the 'control' element of participating overlay plug-ins by factoring out the task of overlay maintenance and delegating this to the framework. We also have a PhD project that is using the open overlays framework as the basis of a sub-framework specialising in ad-hoc routing protocols in MANETs.

In future work, we are particularly interested in supporting challenging scenarios involving 'extreme' network heterogeneity of the type discussed in the introduction (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 DCFs 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.).

More widely, we believe that users of mainstream middleware will increasingly demand overlay support, and so the challenge will arise of how best to integrate the open overlays concept, or a variant of it, in such platforms. We hope that the experiences reported in this paper will be of relevance in this.

7. ACKNOWLEDGEMENTS

This work has been carried out in the EPSRC-funded Open Overlays project (GR/S68521/01) and the NWDA-funded NW Grid project. We would also like to acknowledge the valuable contributions of several colleagues including Gareth Tyson, Chris Cooper, David Duce, Musbah Sager, Wei Li, Laurent Mathy and Wei Cai.

8. REFERENCES

[1] A. Andersen, G. S. Blair, V. Goebel, R. Karlsen, T. Stabell-Kul and W. Yu. Arctic Beans: Configurable and Reconfigurable Enterprise Component Architectures. *IEEE Distributed Systems Online*, 2 (7), November 2001.

[2] S. Behnel and A. Buchmann. Overlay Networks - Implementation by Specification. *In Proceedings of the ACM/IFIP/Usenix International Middleware Conference*, pages 401-410, Grenoble, France, November 2005.

[3] K. Beven, R. Romanowicz, F. Pappenberger, P. Young and M. Werner. The Uncertainty Cascade in Flood Forecasting. *In Proceedings of the ACTIF meeting on Flood Risk*, Tromsø, Norway, October 2005.

[4] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas and K. Saikoski. The Design and Implementation of Open ORB V2. *IEEE Distributed Systems Online*, 2(6), September 2001.

[5] D. Bromberg and V. Issarny. INDISS: Interoperable Discovery System for Networked Services. *In Proceedings of the ACM/IFIP/Usenix International Middleware Conference*, pages 164-183, Greoble, France November 2005.

[6] L. Capra, W. Emmerich and C. Mascolo. CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29(10):929-945, October 2003.

[7] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A Large-scale and Decentralized Application-level Multicast Infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC*), 20(8):1489–1499, October 2002.

[8] B. Cooper. Trading off Resources Between Overlapping Overlays. *In Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference*, pages 101-120 Melbourne, Australia, December 2006.

[9] C. Cortes, G. Blair and P. Grace. A Multi-protocol Framework for Ad-hoc Service Discovery. *In Proceedings of the 4th International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC '06)*, Melbourne, Australia, November 2006.

[10] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. Picco, T. Sivaharan, N. Weerasinghe and S. Zachariadis. The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario. *In Proceedings of the. 5th IEEE International Conference on Pervasive Computing and Communications (PERCOM'07)*, pages 69-78, White Plains, New York, March 2007.

[11] G. Coulson, G. Blair, D. Hutchison, A. Joolia, K. Lee, J. Ueyama, A. Gomes and Y. Ye. NETKIT: A Software Component-Based Approach to Programmable Networking. *ACM SIGCOMM CCR*, 33(5): 55-66, October 2003.

[12] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, K., and J. Ueyama, J. A Component Model for Building Systems Software. *In Proceedings of Software Engineering and Applications (SEA'04*), Cambridge, MA, USA, ACTA Press, ISBN 0-88986-425-X, November 2004.

[13] G. Coulson, D. Kuo and J.Brooke. Sensor Networks + Grid Computing = A New Challenge for the Grid? *IEEE Distributed Systems Online*, 7(12), http://dsonline.computer.org/portal/pages/dsonline/2006/12/o12002.html, December 2006.

[14] G. Cugola, A. Murphy and G. Picco. Content-Based Publish-Subscribe in a Mobile Environment. *Handbook of Mobile Middleware*, Corradi, A., and Bellavista, P. eds., pages 257-285, Auerbach Publications, 2006.

[15] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz and I. Stoica. Towards a Common API for Structured P2P Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 33–44, Berkeley, CA, USA, February 2003.

[16] N. Davies, A. Friday, S. Wade and G. Blair. L^2imbo: A Distributed Systems Platform for Mobile Computing. *Mobile Networking Applications*, 3(2):143-156, August 1998.

[17] V. Eide, F. Eliassen and O. Lysne. Supporting Distributed Processing of Time-based Media Streams. *In Proceedings of Distributed Objects and Applications (DOA'01*), pages 281-288, IEEE, Rome, Italy, September 2001.

[18] M. Fleury and F. Reverbel. The JBoss Extensible Server. *In Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, pages 344-373, Rio, Brazil, June 2003.

[19] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *In Proceedings of the IFIP International Conference on Network and Parallel Computing*, Springer-Verlag LNCS 3779, pages 2-13, 2006.

[20] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Parallel Computing*, 28(12):1753–1772, December 2002.

[21] A. Ganesh, A. Kermarrec and L. Massoulie. SCAMP: Peer-to-peer lightweight membership service for large-scale group communication, *In Proceedings of the 3rd Int.Workshop on Networked Group Communication*, London, UK, 2001.

[22] P. Grace, G. Blair and S. Samuel. ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, pages 1170-1187, Catania, Sicily, Italy, November 2003.

[23] P. Grace, G. Coulson, G. Blair, L. Mathy, W. Yeung, W. Cai, D. Duce, and C. Cooper. GridKit: Pluggable Overlay Networks for Grid Computing. *In Proceedings of the International Symposium on Distributed Objects and Application*s, pages 1463-1481, Cyprus, October 2004.

[24] P. Grace, G. Coulson, G. Blair and B. Porter. Deep Middleware for the Divergent Grid. *In Proceedings of the 6th IFIP/ACM/USENIX International Middleware Conference*, pages 334-353, Grenoble, France, November 2005.

[25] P. Grace, G. Coulson, G. Blair and B. Porter. A Distributed Architecture Meta Model for Self-Managed Middleware. *In Proceedings of the 5th International Workshop on Adaptive and Reflective Middleware (ARM '06)*, co-located with Middleware 2006, Melbourne, Australia, November 2006.

[26] K. Henricksen and R. Robinson. A Survey of Middleware for Sensor Networks: State-of-the-art and Future Directions. *In Proceedings of the International Workshop on Middleware For Sensor Networks*, Melbourne, Australia, November 06, MidSens '06, Vol. 218, ACM Press, 2006.

[27] V. Hingne, A. Joshi, T. Finin, H. Kargupta and E. Houstis. Towards a Pervasive Grid. *In Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IPDPS*, Nice, France, April 2003.

[28] D. Hughes, P. Greenwood, G. Coulson, G. Blair, F. Pappenberger, P. Smith and K. Beven. An Intelligent and Adaptable Flood Monitoring and Warning System. *In Proceedings of the 5th UK E-Science All Hands Meeting (AHM'06)*, Nottingham, UK, September 2006 http://www.allhands.org.uk/2006/proceedings/proceedings/.

[29] A. Joseph, A. de Lespinasse, J. Tauber, D. Gifford and M. Kaashoek. Rover: a Toolkit for Mobile Information Access. *In Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, United States, Jones, ed., SOSP '95, pages 156-171, ACM Press, December 1995.

[30] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. Magalhães and R. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. *In Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000*), pages 121-143, New York, NY, USA, April 2000.

[31] B. Lagaisse and W. Joosen. True and Transparent Distributed Composition of Aspect-Components. *In Proceedings of the International ACM/IFIP/Usenix Middleware Conference*, LNCS 4290, pages 42-61, Melbourne, December 2006.

[32] B. Li, J. Guo and M. Wan. iOverlay: A Lightweight Middleware Infrastructure for Overlay Application Implementations. *In Proceedings of the ACM/IFIP/USENIX International Middleware Conference* (Middleware 2004), pages 135-154, Toronto, Canada, October 2004.

[33] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe and I. Stoica. Implementing Declarative Overlays. *SIGOPS OSR*, 39(5):75-90, 2005.

[34] B. Maniymaran, M Bertier, and A. Kermarrec. Build One, Get One Free: Leveraging the Coexistence of Multiple P2P Overlay Networks. *In Proceedings of ICDCS 2007*, Toronto, Canada, June 2007.

[35] L. Mathy, R. Canonico and D. Hutchinson. An Overlay Tree Building Control Protocol. *In Proceedings of the 3rd International COST264 Workshop on Networked Group Communication*, pages 76–87, London, UK, November 2001.

[36] B. Porter, F. Taiani and G. Coulson. Generalised Repair for Overlay Networks. *In Proceedings of International Symposium on Reliable Distributed Systems (SRDS 2006)*, pages 132-142, Leeds, UK, October 2006.

[37] O. Riva. Contory: A Middleware for the Provisioning of Context Information on Smart Phones. *In Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, pages 219-239, Melbourne, Australia, December 2006.

[38] A. Rodriguez, C. Killian, S. Bhat, D. Kostic and A. Vahdat. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. *In Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI2004)*, pages 267-280, San Francisco, CA, USA, March 2004.

[39] M. Roman, D. Mickunas, F. Kon and R. Campbell. LegORB and Ubiquitous CORBA. *In Proceedings of the Workshop on Reflective Middleware, IFIP/ACM Middleware'2000*, IBM Palisades Executive Conference Center, NY, April 2000.

[40] M. Roman and N. Islam. Dynamically Programmable and Reconfigurable Middleware Services. *In Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 372–396, Toronto, Canada, November 2004.

[41] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. *In Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*, pages 329-350, Heidelberg, Germany November, 2001.

[42] S. Sadjadi, P. McKinley and E. Kasten. Architecture and Operation of an Adaptable Communication Substrate. *In Proceedings of the 9th IEEE International Workshop on Future Trends of Distributed Computing Systems* (FTDCS'03), pages 46–55, San Juan, Puerto Rico, May 2003.

[43] A. Sage and C. Cuppan. On the Systems Engineering and Management of Systems of Systems and Federations of Systems. *Information, Knowledge, Systems Management*, 2(4): 325-345, 2001.

[44] I. Stoica, R. Morris, R. Karger, M. Kaashoek and H. Balakarishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications, *In Proceedings of ACM SIG-COMM*, pages 149-160 San Diego, August 2001.

[45] R. van Renesse, Y. Minsky and M. Hayden. A Gossip-Based Failure Detection Service. *In Proceedings of the 1st IFIP International Conference on Middleware*, pages 55–70, Lake District, UK, September 1998.

[46] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Adaptive Systems Using Ensemble. *Software Practice and Experience*, 28(9): 963–979, August 1998.

[47] G. Xiaohui and K. Nahrstedt. An Event-Driven, User-Centric, QoS-aware Middleware Framework for Ubiquitous Multimedia Applications. *In Proceedings of 9th ACM Multimedia (Multimedia Middleware Workshop*), Ottawa, Canada, October 2001.

[48] C. Zhang and H. Jacobsen. Refactoring Middleware with Aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1058-1073, November 2003.