Deep Middleware for the Divergent Grid

Paul Grace, Geoff Coulson, Gordon S. Blair, and Barry Porter

Computing Department, Lancaster University, Lancaster, UK
{gracep, geoff, gordon, porterbf}@comp.lancs.ac.uk

Abstract. Next-generation Grid applications will be highly heterogeneous in nature, will run on many types of computer and device, will operate within and across many heterogeneous network types, and must be explicitly configurable and runtime reconfigurable. We refer to this future Grid environment as the "divergent Grid". In this paper, we propose a "deep middleware" approach to meeting key requirements of the divergent Grid. Deep middleware reaches down into the network to provide highly flexible network support that underpins a rich, extensible and reconfigurable set of application-level "interaction paradigms" (such as publish-subscribe, multicast, tuple spaces etc.). In our Gridkit middleware platform, these facilities are encapsulated in two key component frameworks: the interaction framework and the overlay framework, which are the subject of this paper. The paper also evaluates the two frameworks in terms of their configurability (e.g. ability to be profiled for different device types) and reconfigurability (e.g. to self-optimise as the environment changes).

1 Introduction

As Grid computing continues to evolve, there is an accelerating trend towards diversity both in terms of application domains and, crucially, in terms of the underlying networked infrastructures in use. For example, with the emergence of the "pervasive Grid" [11], we can envisage a spectrum ranging from very large cluster computers interconnected with high-speed networks through to tiny embedded devices interconnected by often intermittent and low bandwidth wireless networks.

A more detailed analysis of heterogeneity at the infrastructure level of the Grid reveals the following:

- *At the network level.* Beginning with dedicated intra-cluster networking, the range of network types in use has grown to include: high-speed local networks; lower-speed wide-area networks; infrastructure-based wireless networks; adhoc wireless networks (themselves ranging from relatively static to highly dynamic configurations); and specialised sensor networks.
- *At the middleware level.* Beginning with basic point-to-point interactions (e.g. SOAP messaging and RPC), the range of middleware-level communications services in use is expanding to encompass a wide range of "interaction paradigms" such as: reliable and unreliable multicast; workflow; media streaming; publish-subscribe; generative communication; and peer-to-peer based resource location or file sharing.

G. Alonso (Ed.): Middleware 2005, LNCS 3790, pp. 336–355, 2005.
© IFIP International Federation for Information Processing 2005

We characterise these trends as the *divergent Grid*. As a more concrete illustration of the divergent Grid, consider the following scenario which is currently being realised at Lancaster University [15]: *A river and estuary are instrumented with a range of sensor devices e.g. to monitor temperature, water levels, flow rates, pollution levels, coastal erosion etc. Some of these devices (e.g. fixed sensors in tidal defence walls) are networked using standard wired technologies such as Ethernet, while others employ various wireless technologies (e.g. IEEE 802.15.4 or 802.11 radios; or longwave radios for underwater use). Using this infrastructure, scientists in widely-dispersed locations selectively store sensor data for future analysis, integrate and process live sensor data on their workstations, cooperatively visualise this data in real-time (supported by a video conferencing system), and use both stored and live data to computationally steer long running environmental simulations on computational clusters.*

Note that this divergent Grid scenario clearly involves highly heterogeneous device and networking technologies, and also that it demands a wide range of interaction paradigms (e.g. ad-hoc multicast for sensor data dissemination, publish-subscribe for sensor data collection, multicast and streaming for collaboration, and secure channels for database access). Dealing with such extreme heterogeneity is a fundamental challenge for future Grid middleware, and one that is demonstrably not addressed by existing platforms (as is also argued in [7]). In this paper, we propose a platform called Gridkit that tries to address these deficiencies. Gridkit adopts and builds on our previous approach to the development of reflective middleware [4]: it utilises components, reflection and component frameworks to yield a configurable, reconfigurable and evolvable architecture.

But the most novel contribution of Gridkit is that it explores the notion of *deep middleware* in which the middleware platform reaches down into the (heterogeneous) network to provide flexible communications services with which to support a range of distributed interaction paradigms at the application level. Deep middleware can either build on support from an active or programmable network, or can leverage the notion of *overlay networks* [12]. In our previous work [9] we have explored the former; in the present work we explore an overlay-based approach which has the key advantage that it can be applied in 'black box' network environments.

In outline, Gridkit has at its heart two layered component frameworks. The higher layer is an *interaction framework* that takes plug-in interaction paradigms; the lower layer is an *overlay framework* which takes plug-in overlay implementations. See figure 1. In previous work [15], we have provided an outline of the wider Gridkit architecture which supports an API based on web-services and also in-

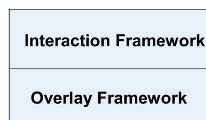

Fig. 1. The overall architecture

cludes frameworks for Grid-based resource discovery, service discovery, resource management, and security. In this paper we focus on the 'heart' of Gridkit: the above-mentioned interaction and overlay frameworks. In particular, we demonstrate how the deep middleware approach can support a rich, extensible and reconfigurable set of application-level interaction paradigms in and across a variety of network types and on a variety of devices.

The remainder of the paper is structured as follows. Sections 2 and 3 respectively discuss the interaction and overlay frameworks. Section 4 then presents a study of the configurability of the two frameworks (how they can be instantiated on different device types) and their reconfigurability. The reconfigurability study focuses especially on self-managing functionality offered by the overlay framework. Following this, we discuss related work in section 5 and present our conclusions and plans for future work in section 6.

2 The Interaction Framework

2.1 Motivation

Grid Middleware that offers only a single interaction paradigm (e.g. RPC) cannot cope with the diversity of application requirements needed by next-generation Grid applications [7]. This is illustrated clearly in the environmental informatics scenario of section 1 which, as explained, involves at least publish-subscribe, multicast-based group interaction and media streaming.

One possible solution to this problem is to *employ separate middleware implementations* for each interaction paradigm required. This solution is implicit 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:

- being responsible for middleware composition and integration adds considerable complexity to the load on the application developer;
- it is unlikely that all implementations of the same interaction paradigm will support the same programming model, programming language and operation syntax, which further increases the cognitive load on the developer;
- 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 interaction paradigm implementations may only operate in certain environments and/or under certain network conditions (e.g. different publish subscribe implementations are typically used for infrastructure-based and for ad-hoc networks) this again leads to redundant deployment, this time of individual interaction paradigms.

2.2 Overview of the Interaction Framework

To address these problems, Gridkit's interaction framework provides a common environment for an extensible set of so-called pluggable interaction paradigms, or PIPs.

The design of the 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 over different (overlay) network types and conditions;
3. the configuration of PIPs, including their underlying overlay support, should be managed automatically based on an (optional) declarative specification of desired behaviour;
4. the configuration of PIPs should also be informed by the currently available network infrastructure and environmental conditions.

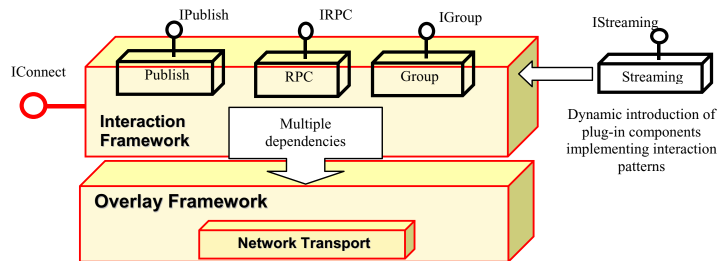

Fig. 2. The interaction framework

The overall architecture and context of the interaction framework is illustrated in figure 2. Separating the interaction framework from the overlay framework has the effect of promoting the reuse of overlays and thus conserving 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 - see section 3.2). Additionally, figure 2 shows that a *network transport framework* is plugged into the overlay network framework; this provides components (e.g. TCP, UDP etc.) that implement communications services that are used directly by overlays, and that are used directly by PIPs that do not require sophisticated overlay support (e.g. RPC).

The interaction framework does not impose any specific structure on its plug-ins except that it requires that each plug-in is encapsulated as a single component. However, as our OpenCOM v2 component model [8] supports composite components, this imposes no real constraint.

2.3 Interaction Framework APIs

In line with principles 1 and 2 set out 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 its

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 are likely to 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.

Because of the variety of interaction paradigms and the need to support future extensibility, it is unrealistic to define universal, fixed, interfaces to PIPs. Instead, we adopt an approach to API provision that relies on the definition of an (extensible) set of *generic APIs*. The expectation is that each generic API will be exported by a potentially large family of underlying PIPs. In cases where a PIP requires a modification of the generic API closest to its needs, the framework recommends that interface inheritance be used wherever possible to avoid a proliferation of top-level APIs. Avoiding a proliferation of top-level APIs is crucial in giving applications some level of stability and consistency, and in enabling them to accumulate transferable knowledge. As an example, a new group communication PIP that addresses message ordering issues could not directly use a group API that is silent on message ordering. However, the PIP developer should extend this generic API rather than add an entirely new one.

In addition to providing recommendations for the structuring of PIP APIs, we have attempted to simplify the way in which applications *select and configure* PIPs. Our approach here employs a notion of so-called *binding contracts* that is in turn inspired by the idea of 'trading' in RM-ODP [23]. 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 (a receptacle is a 'required' interface [8]) of application components that want to use PIPs have *predicates* attached to them whose terms refer to the name-value pairs attached to potentially-matching PIP interfaces. The binding contract elements (i.e. name-value pairs and predicates) are attached to receptacles and interfaces using native facilities of our component model (i.e. the 'interface' meta-model as described in [3]).

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. Given this, the interaction framework selects, instantiates, and configures a PIP instance based on the following information:

- the set of available PIPs that are currently registered with the framework;
- the predicates attached to the offered receptacle;
- the advice of a *context engine* [5] 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.)

During the process of finding a suitable PIP, the predicate attached to the user's receptacle must evaluate to *true* when bound to the name-value pairs from both the selected PIP interface and the context engine. Section 4 has specific examples of the use of binding contracts and related machinery.

Additionally, the interaction framework (optionally) supports *dynamic monitoring* of binding contracts. Using this facility, any 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 can drive reconfiguration of a PIP (e.g. by reconfiguring its underlying overlay stack; see section 3) by altering the predicates attached to its receptacle. To detect such changes, the component model's 'interception' meta-model [3] is used to attach a 'dynamic contract evaluator' to the receptacle-interface binding. This is executed each 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 or by the framework itself, e.g., to delete the PIP instance or to 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 flooding to an ad-hoc network based flooding. Again, see section 4 for examples and more detail.

3 The Overlay Framework

3.1 Background on Overlays

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. They are typically implemented by deploying appropriate application-level routing functionality at strategic places in the network (in principle both at the network edges and in the core). Overlays have to date mainly been motivated by two concerns: i) to alleviate the effects of slow or sporadic deployment of new services in the Internet (e.g. application-level multicast); and ii) to directly provide application-level functionality that is out-of-scope for the underlying network (e.g. large-scale peer-to-peer file sharing). Examples of overlay types are: reliable multicast overlays such as SRM; content dissemination networks; unstructured peer-to- peer overlays such as Gnutella; structured dynamic hashtable (DHT)-based peer-to-peer overlays such as Chord; resilient overlay networks (RONs); gossip overlays; and the wide variety of routing overlays used in ad-hoc or wireless sensor networks. See [15] for a survey.

3.2 Overview of the Overlay Framework

Gridkit's overlay framework supports the design, deployment and management of plug-in overlay networks. In terms of design, the framework mandates that per-host overlay plug-ins are structured in terms of three standard elements (components). These (see figure 3) are: i) a *control component* that cooperates with its peers on other hosts to build and maintain a virtual network topology, ii) a *forwarding component* that routes messages over the virtual topology, and iii) a *state component* that encapsulates key state such as nearest neighbours. This tri-partite structure provides a useful pattern for developers, promotes the

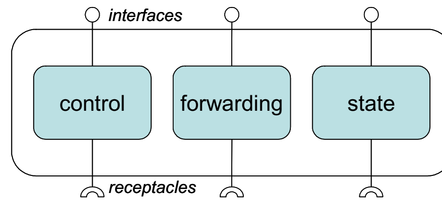

Fig. 3. Structure of an overlay plug-in

dissemination of experience and expertise in overlay development, and facilitates deployment and management. Note also in figure 3 that each of the 3 elements exposes an interface to the higher layer and a receptacle to the lower layer.

In terms of *deployment*, the overlay framework allows one to dynamically instantiate new overlays in a straightforward and lightweight manner. This is supported in a recursive fashion by using overlays to deploy overlays (PIPs are also deployed in this way). For example, a flooding-based overlay (e.g. Gnutella [14]) can be used to disseminate a message that (a filtered subset of) receiving hosts act upon by deploying a node of a new overlay of some desired type (e.g. an application-level multicast overlay). This is achieved by employing a *stack structure* for overlay implementations, and adopting an associated message handling regime that is inspired by the Ensemble communications framework [29]. In brief, the forwarding elements of overlays are organised such that when an incoming message is not recognised, it is passed up to the forwarding component of the overlay above. Given this arrangement, one can place a 'dummy' overlay at the top of the overlay stack that responds to deployment request messages. Such requests will necessarily reach the top of the stack as they will not have been recognised by any of the lower forwarding components.

Apart from its use in deployment, the general notion of stacking overlays is a powerful one, and there are numerous cases in which one overlay can usefully be employed as a substrate for another. For example, one could layer a keyword search overlay such as Gnutella over a DHT-based network such as Chord (as DHT networks do not support keyword search). Or, one could layer a content dissemination overlay such as TBCP [21] over a resilient overlay such as RON [2] to enhance dependability. All such scenarios can be achieved very easily using the overlay framework's stacking structure.

As well as stacking whole overlays, the overlay framework also supports *partial stacking* in which the control, forwarding, and state elements can be separately stacked. For example, we have designed a variant of Gnutella [17] that builds a more structured network than the completely unstructured topology constructed by standard Gnutella. This variant can be deployed simply as a <*control, state*> pair, and an existing standard Gnutella forwarding component in the layer below can be used directly. Another example of partial stacking could be the stacking of a multicast overlay over a DHT-based overlay. Here, the multicast overlay would only need to provide a forwarding component, as the control and state components of the underlying DHT overlay could be used directly.

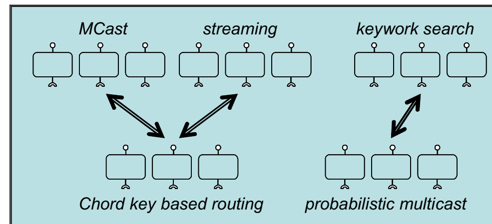

Fig. 4. Example configuration of the overlay framework

Partial stacking not only saves developer effort it also potentially conserves resources, as functionality common to a set of stacked overlays can be reused, thus saving end-system resources and potentially reducing network traffic.

Figure 4 illustrates an example configuration of the overlay framework that involves two multi-layered overlay instantiations: first, a group overlay and streaming overlay are both supported by an instance of the Chord DHT overlay; second, a keyword search overlay is supported by a probabilistic multicast overlay. This demonstrates how multiple overlay networks, both related and unrelated, can co-exist within a single middleware platform instance; and how overlays can be configured on top of other overlays to construct higher-level, more application-specific semantics.

As well as stacking, the overlay framework also promotes horizontal cooperation between different overlays. For example, as explored in section 4, a gossip-based overlay can be used to gossip about crashed nodes in a different overlay, and thus be used to provide a general failure detection service for other overlays. Similarly, an overlay that provides a dependability service for the nodes of other overlays could exploit a third overlay to search for suitable hosts on which overlay nodes could be redundantly checkpointed. As a third example, separate infrastructure-based and ad-hoc-based multicast overlays could cooperate side-by-side to underpin a publish subscribe PIP that must simultaneously operate in both network environments.

Finally, in terms of the *management* of deployed overlays, the overlay framework employs plug-in ‘component configurators’ [19] that builds on another of the component model’s reflective meta-models - this time the ‘architecture’ meta-model [3]. But in addition, some management functions can be carried out by overlays themselves. Within a single overlay, it is the responsibility of the control part of the implementation to manage, maintain, and repair the overlay topology. But it is also possible to use specialised overlays to manage other overlays. Examples of this relating to failure detection and dependability have already been given above and are pursued in section 4.

3.3 Overlay Framework APIs

The general approach of interfacing users to the overlay framework is identical to that adopted by the interaction framework (see section 2.3): viz. the convention of an extensible set of generic APIs that can each support a family of related

Table 1. Generic overlay APIs

	DHT	Cast
Control	*join(networkId)* *leave(networkId)*	*join(grpId)* *leave(grpId)*
Forwarding	*put (key, data)* *remove (key)* *value = get (key)*	*multicast(msg, grpId)* *anycast(msg, grpId)*
State	*nodes = neighbours()* *addneighbour(node)*	*nodes = neighbours()* *addneighbour(node)*

underlying overlays. In addition, the framework uses the *'connect()'* API and binding contracts to select, configure and dynamically monitor overlays.

Our current set of generic APIs, which are taken almost directly from [10] except that they are factored into control, forwarding and state categories, is shown in table 1. This shows two generic APIs for DHT-based and for cast-based overlays respectively. Following Dabek et al's experience we have found that these generic APIs can be used by a large family of overlay plug-ins. For example, the generic DHT API can give access to Chord, Pastry, Tapestry etc., and the cast API can give access to multicast overlays, ad-hoc routing protocols etc. The complete set of overlays that we have implemented is listed in section 6.

Finally, note that in the case of the overlay framework, the *'connect()'* process naturally recurses to drive the instantiation of stacks of overlays: i.e., if the initial *connect()* call instantiates a new overlay plug-in, the instantiation of this might in turn drive the instantiation of another below it. And so on.

4 Case Studies of Configuration and Reconfiguration

4.1 Configuration

In this section we demonstrate the *configurability* of Gridkit on different computer and device types, showing how different PIPs can be automatically configured and underpinned with overlay configurations in a way that is appropriate to different environmental conditions. In particular, we discuss scenarios in which we configure two different types of PIP on two different types of device: a PC and a PDA. We also concretise the discussion on binding contracts in section 2.3 by giving examples of the use of binding contracts and their associated machinery.

Consider a Gridkit installation that is described by table 2. 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 we are considering. 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 services; and *Net* means network type (i.e. fixed or ad-hoc).

Given this installation, consider the processing of a request on the interaction framework of the form *connect(publish-receptacle)* for an IPublish generic API

Table 2. An example Gridkit installation

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

where there is a predicate of the form *RelMes=F* attached to *publish-receptacle*. The steps involved in processing this request are as follows (please refer to figure 5):

- Step 1: the *connect(publish-receptacle)* call is issued by the application on the interaction framework as already described.
- Step 2: the interaction framework picks a PIP that exports the specified generic API, and retrieves from the context engine the set of contextual name-value pairs that are relevant to the type of this PIP - in this case it picks *Publish* and retrieves *Net: fixed* if running on a PC, or *Net: adhoc* if running on a PDA (the name-value pairs deemed relevant for a given PIP are designated by the PIP developer when the PIP is first registered with the framework).
- Step 3: a pattern-matching algorithm (similar to that used in [5]) is used to select a per-PIP 'configuration script' on the basis of the receptacle predicate and the name-value pairs from the context engine and from candidate PIPs (again, this configuration script is provided when the PIP is first registered).
- Step 4: the script instantiates the PIP and then decides on a suitable overlay type to underpin the PIP; in this case it will pick the *IGroupMessage* generic API underpinned by an Application Level Multicast (*ALM*) implementation [21] on the PC because *ALM's RelMes* and *Net* values satisfy both the publish-receptacle's predicate of *RelMes=F* and the *Net* value provided by the context engine; it will, however, be underpinned by *ProbMcast* on the PDA due to the fact that this exports *Net: adhoc* which matches the Net value exported by the context engine; the script also derives a suitable predicate for the overlay receptacle *alm-receptacle* (in this case the predicate will be *RelMes: F*), and attaches this to the *alm-receptacle*.
- Step 5: the script issues a *connect(alm-receptacle)* call on the overlay framework.

From this point on, steps 6, 7 and 8 are analogous to the steps already described above except that they are executed by the overlay framework rather than the interaction framework. The final results are shown in figure 6. Note that the *connect()* process may be carried out multiple times by the overlay framework in the case of a request that indicates a stack of overlays.

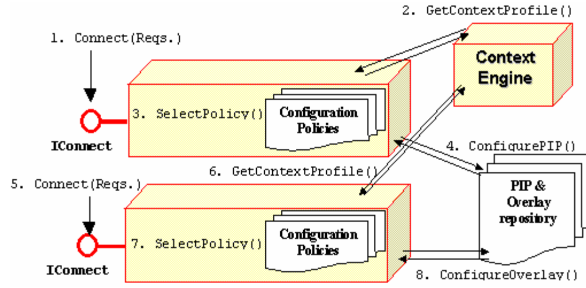

Fig. 5. Steps involved in processing a *connect()* request

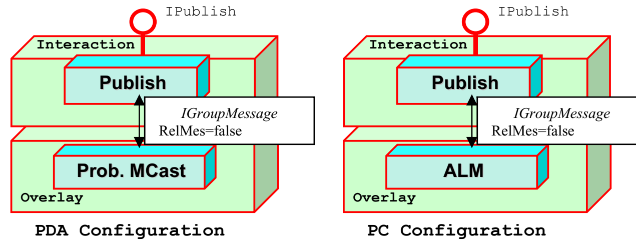

Fig. 6. Applying publish configurations on the PC and the PDA

Now consider a consider a request on the interaction framework 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 (again due to contextual differences). The Gossip overlay is used to gossip about group membership (as required by the *GrpMem=T* predicate). The outcomes are shown in figure 7.

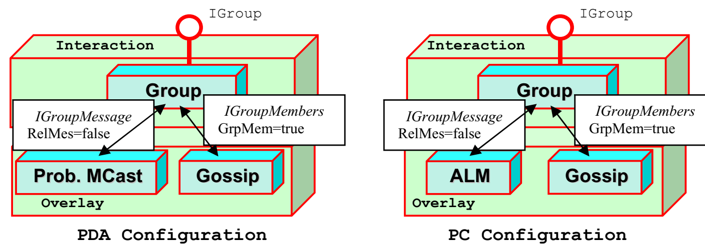

Fig. 7. Applying group configurations on the PC and PDA

Note that the above processes rely on an 'ontology' of names (*RelMes*, *GrpMem* etc) which are commonly understood across the two frameworks and the context engine. Although it leads to a degree of 'coupling' between the frameworks, this is a necessary evil in realising automatic configuration of PIPs/overlays. In general it is not as much of a problem as it might initially seem, as a natural convention emerges under which PIP developers build on a canonical set of names used by the lower level frameworks.

Table 3. Memory footprint sizes of the four configurations

Configuration	Static Memory Footprint (KBytes)	Configuration Time (ms)
Publish-Subscribe with ALM (PC)	171	616
Pub-Subscribe with ProbMcast (PDA)	223	3012
Group with ALM (PC)	221	591
Group with ProbMcast (PDA)	276	3776

Overhead Evaluation. For completeness we briefly present the times taken to generate the above configurations and the memory footprint incurred. These are presented in table 3. The experiments were carried out on the following platforms. The PC was a Dell Optiplex workstation with a 3.0 GHz Pentium 4 processor and 1Gbyte of RAM with a fixed network connection and running Windows XP. The PDA was a Compaq iPaq H360 2002 with a 233Mhz StrongARM processor and 32Mbytes of RAM with an ad-hoc network connection and running Windows Pocket PC. Details of the implementation environment of the frameworks are given in section 6.

4.2 Reconfiguration

We now present a case study that demonstrates one way in which *dynamic reconfiguration* of the overlay framework can benefit the overall performance of Gridkit. As previously described, it is possible to simultaneously support multiple overlays in a single overlay framework configuration. However, there is a potential source of redundancy in multi-overlay configurations in that individual overlays often provide (in their 'control' elements) their own proprietary network monitoring and repair mechanisms which may have overlapping functionality. In this case study, we investigate the potential for dynamically replacing individual overlay monitoring mechanisms with a generic mechanism that can be shared across overlays, thus reducing network messaging overhead.

We consider two overlays, each of which we have re-implemented to fit the requirements of the overlay framework: Chord [28] is a DHT-based overlay that performs key-based routing, and Scribe [6] is a tree-based overlay that performs multicast/anycast routing of messages under different 'topics' atop a keybased routing mechanism (e.g. Chord). In terms of overlay maintenance, Chord nodes

continuously monitor and repair their network structure by sending control messages to their logical neighbours. Similarly, Scribe nodes periodically send 'heartbeats' to their child nodes, and receive heartbeats from their parents. A detected change in either network (due to the arrival of new nodes or node failures) triggers the execution of a proprietary repair algorithm.

The architecture of our Chord and Scribe implementations is illustrated in figure 8. It can first be seen that Scribe is stacked on top of Chord in the manner discussed in section 3.2. The figure also shows two versions of the control elements of each overlay: an *active* and a *passive* version. In each case, the active version encapsulates the overlay's proprietary monitoring and repair algorithm (as described above), whereas in the passive versions we have removed the monitoring aspect of the algorithm and left only the repair aspect. The intention is that the monitoring element, in each case, will be provided by a common monitoring service.

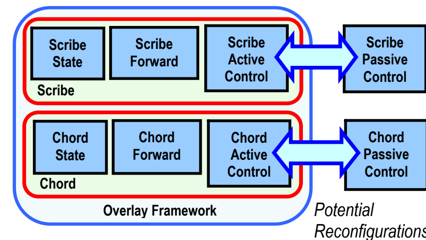

Fig. 8. Chord and Scribe overlays with alternative control elements

Our implementation of this common monitoring service (see figure 9) is based on a gossip failure detection scheme proposed by [31]. The basic operation of each gossip overlay node is to 'gossip' a given message to a specified random subset (K_{gossip}) of its neighbours. On top of this overlay, we have implemented a special-purpose *monitoring overlay*, the nodes of which periodically gossip a heartbeat counter indicating their 'alive' status to local neighbours. Each node monitors heartbeat activity, and if it hasn't received a heartbeat update from a given node in a given time period, it declares the node 'dead'.

In operation, therefore, the intention is that the monitoring and gossip overlays are used to send messages across all nodes in both the Scribe and Chord overlays about fails and joins, and this information is used to replace Scribe's and Chord's proprietary monitoring mechanisms and to drive their passive control elements.

To confirm the benefits (in terms of overall network overhead) of reconfiguring from an active to a passive control strategy, we set up an experimental configuration that involved 10 instances of the overlay framework running on 5 workstations. One of these workstations, designated as the *test host*, was set up to measure the total number of failure detection related control messages originating from that host.

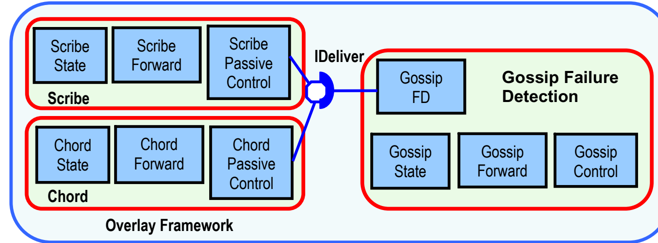

Fig. 9. Configuration of overlay framework with gossip failure detection

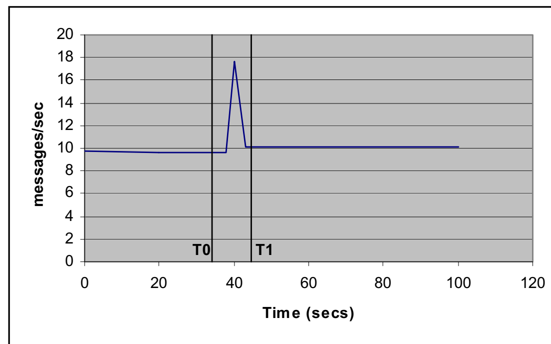

Fig. 10. Investigation of control message throughput in the overlay framework

We then configured the overlay framework to switch from an active to a passive control strategy when a threshold rate of 11 messages/sec of measured control messages was exceeded. Given this set up, we proceeded as follows (please refer to figure 10): We first instantiated a Chord overlay on all the experimental hosts; this produced a control message rate of approximately 10 messages/sec. Then, after 35 seconds (time T_0) we instantiated a two Scribe trees on top of Chord (these were configured so that the nodes on the test host acted as parent to three child nodes in one tree, and one in the other). The Scribe trees produced an additional 8 messages/sec; so at this point the combined number of control messages (18) exceeded the configured threshold (11) and forced the following reconfiguration to occur: i) the gossip and failure detection overlays were instantiated (with a heartbeat/monitoring period of 500ms and a K_{gossip} parameter of 5 neighbours); ii) each active control component was replaced by the corresponding passive version; and iii) the passive control components were connected to the failure detection component. That is, framework configuration changed, at time T_1, from the view of figure 8 to the view of figure 9. Under these conditions, the test host measured a message rate of approximately 10 messages/sec which is a significant reduction of the prior rate of 18. Note also that this rate will remain constant no matter how many overlays share the failure detection service.

Finally, we measured the overhead of the active-to-passive transition, broken down into discrete phases. It can be seen from table 4 that there was a total overhead (downtime) of 1.7 seconds while the transition is taking place; during this time any PIPs that are using the framework would be blocked. However, this is an 'out-of-band' operation that occurs only once, and once completed does not further impact the performance of the middleware.

Table 4. Time to reconfigure the overlay framework

Operation	Time(ms)
Configure Gossip Failure Detection	547
Replace 2 active control components	94
Connect FD to running overlays (Java to C++ bridge)	1141
Total Time	1782

It is important to emphasise that this failure detection approach cannot be applied under all circumstances. In particular, it is only applicable to overlays that are 'fully connected' in the sense that it is possible to reach all nodes from any given node. This property is required to be able to deploy the Gossip overlay according to the scheme outlined in section 3.2 on all nodes that require monitoring. Also, it might not necessarily be the case that the Gossip approach leads to overlays being repaired as quickly and/ or effectively as their proprietary mechanisms achieve. Nevertheless it is a clear illustrative example of the potential benefits of 'horizontally' composing overlays which is facilitated by our framework.

5 Related Work

We are not aware of any other work that is specifically addressing the provision of integrated support for pluggable overlay networks or interaction paradigms in Grid environments. However, there is a considerable amount of related work in the various sub-areas.

In terms of Grid middleware, there are platforms, notably ICENI [13], that support the notion of software components. However, these platforms, so far as we are aware, support components only at the application level: there is no infrastructure level componentisation. In terms of wider, non-Grid-specific, middleware, there are many platforms that take a component-oriented approach at the infrastructure level, and feature plug-ins to extend system functionality. Among these are DynamicTAO [19], UIC [26], ExORB [25], Arctic Beans [1] and RAPIDware [27]. However, none of these support the notion of pluggable interaction paradigms or overlay networks.

There is, of course, considerable research in the narrower field of overlay networks themselves; but this work is largely orthogonal to our focus: we are interesting in wrapping and applying overlay technologies rather than in developing new ones. In terms of generic support platforms for overlays, researchers

at the University of Toronto have developed a generic platform called *iOverlays* [20] that supports the implementation of overlays. Essentially, iOverlays is a low-level software cross-connect that forwards messages according to a script that embodies the semantics of a particular overlay. It is thus orthogonal to our interests. Our work also differs in focusing more on co-existence of, and cooperation across, multiple overlay instances which is required to simultaneously support multiple PIPs in the same application. Also in the field of generic overlay support, [10] has presented APIs for common overlay services such as distributed key-based routing, distributed hashtables, distributed object lookup and multicast behaviour. Such APIs offer the potential to simplify the development of distributed systems based upon re-usable overlay services. This is a novel approach that has influenced the design of our overlay framework (see section 3.3). However, we believe that this approach does not go far enough; it concentrates on DHT-based technologies and does not generalise to the many types of overlays that are available (as discussed above). Also, it assumes static layering of overlay types in contrast to our dynamic approach. Hence, we propose a more general approach whereby overlay networks can *arbitrarily* (albeit sensibly) depend upon one another. For example, a publish-subscribe overlay can be layered atop a DHT in one configuration, or a flooding-based overlay in another (e.g. in a small scale ad hoc or wireless sensor network).

Parvalantzas et al. [24] has previously investigated middleware with extensible PIPs (then referred to as *binding types*), and this work has been an influence. However, the present research fundamentally extends this earlier work. In particular, it builds on the availability of the overlay framework to considerably extend the richness and scope of the PIPs that can be provided (e.g. into areas of resource discovery, peer-to-peer file sharing, efficient wide area publish-subscribe, wide area multicast etc). Furthermore, we now accommodate alternative, per infrastructure, PIP implementations, together with their runtime reconfiguration, and also simultaneously support multiple PIPs. We also introduce new mechanisms to support the developer in selecting, configuring and using PIPs.

Finally, there are now a number of established frameworks that support the configuration and reconfiguration of pluggable network protocols. As we have discussed previously, the design of the Gridkit framework is built upon this earlier work; message dissemination through the framework is similar to the Cactus approach [16] i.e. a message is forwarded to interested components only; and the top-level configurator is derived from the Ensemble approach [29]. Hence, with Gridkit we do not present a new approach for the development of such frameworks, rather we apply the concept of pluggable frameworks: across a diverse set of middleware services, in heterogeneous devices and environments. Hierarchical frameworks such as Ensemble [29], Horus [30], and x-kernel [18] provide pluggable stack structures into which micro-protocols implementing smaller protocol functionality are plugged. These systems generally support a single interaction type (normally group communication), and the fine-grained nature of the micro-protocol functionality makes meaningful configuration and reconfiguration of protocols a complex task. Gridkit supports both coarse and fine-grained

reconfigurability, and offers declarative methods to define configurations and reconfigurations. Cactus [16] is the closest framework to Gridkit in terms of its structure and dissemination of messages through the framework; however, it does not consider the potential benefits of dynamically reconfigurable interaction types nor does it examine the benefits of supporting middleware services with overlay networks. Finally, two alternative systems in this area are Appia and SAMOA. Appia [22] supports the co-ordination of multiple channels (related to a common task) operating within the protocol stack; and SAMOA[32] examines support for the concurrent execution of events across micro-protocols in the framework. Neither of these features are addressed in our current Gridkit implementation, and offer potential areas of future research.

6 Conclusions and Future Work

In this paper we have discussed two complementary component frameworks that respectively 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 and to be reconfigured at runtime. The combination thus addresses both of the major requirements of the “divergent Grid” as discussed in the introduction.

To date we have implemented the two frameworks and populated them with a substantial set of plug-ins. In the interaction framework, we have implemented the publish-subscribe and group PIPs that are discussed in this paper in C++ and Java respectively. This multi-language integration is a property of the OpenCOM v2 component model [8] which we use to structure all our software. We have also implemented IIOP and SOAP-based RPC PIPs (in C++) and a streaming PIP (in Java). In terms of overlay plug-ins, Chord, Scribe and Application Level Multicast (i.e. TBCP [21]) have been implemented in Java, and Gossip and ProbMcast have been implemented in C++. The two frameworks themselves, plus the context engine, are implemented in Java. Mostly, we have used the multi-language integration feature for practical reasons to more easily accommodate already-written software into the frameworks.

Although we have made considerable progress, a lot remains to be done. We have addressed dynamic deployment of both overlays and PIPs according to the approach discussed in section 3.2, and we have experimented with reconfiguration. But there is a lot more territory to explore in the area of distributed reconfiguration as discussed in section 4.2. Also, there are a lot of interesting issues in *cross-layer* distributed reconfiguration that involves intelligent cross-coordinated reconfiguration of both frameworks. For example, a publish-subscribe PIP might be adequately underpinned by a TBCP overlay while most or all of its users are situated in the fixed network; but if the situation evolves so that at some point a significant number of users are situated in ad-hoc network environments, then the optimal underpinning of the PIP needs to be reconsidered and should ideally be supported by a coordinated federation of horizontally-composed overlays.

Additional areas of challenge that we are addressing in a follow-on project are the use of Model Driven Architecture to configure our frameworks and also to provide constraint on their reconfiguration; and the use of autonomic techniques so that the frameworks can not only adapt themselves to changing environmental conditions but can also learn from prior adaptations and make better decisions on that basis.

Acknowledgements

This work is funded by the EPSRC under the Open Overlays project (grant reference GR/S68521/01). The authors would also like to acknowledge our colleagues on the project: Chris Cooper, David Duce, Musbah Sager, Wei Li, Laurent Mathy, Wei Cai and Wai-Kit Yeung.

References

1. A. Andersen, G. 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. D. Andersen, H. Balakrishnan, M. Kaashoek, and R. Morris. The Case for Resilient Overlay Networks. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 152–157, Elmau, Germany, May 2001.
3. 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 2. *IEEE Distributed Systems Online*, 2(6), September 2001.
4. G. Blair, G. Coulson, and P. Grace. Research Directions in Reflective Middleware: the Lancaster Experience. In *Proceedings of the 3rd Workshop on Reflective and Adaptive Middleware (RM2004)*, pages 262–267, Toronto, Canada, October 2004.
5. 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.
6. 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.
7. J. Chin and P.V. Coveney. Towards Tractable Toolkits for the Grid: a Plea for Lightweight, Usable Middleware. RealityGrid NeSC Tech Report UKeS-2004-01, February 2004.
8. G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A Component Model for Building Systems Software. In *Proceedings of the IASTED Conference on Software Engineering and Applications (SEA'04)*, Cambridge, MA, USA, November 2004.
9. G. Coulson, G. Blair, D. Hutchison, A. Joolia, K. Lee, J. Ueyama, A.T. Gomes, and Y. Ye. NETKIT: A Software Component-Based Approach to Programmable Networking. *ACM SIGCOMM Computer Communications Review (CCR)*, 33(5):55–66, October 2003.

10. 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.
11. N. Davies, A. Friday, and O. Storz. Exploring the Grid's Potential for Ubiquitous Computing. *IEEE Pervasive Computing*, 3(2):74–75, April-June 2004. see also: http://ubigrid.lancs.ac.uk.
12. D. Doval and D. OMahony. Overlay Networks: A Scalable Alternative for P2P. *IEEE Internet Computing*, 7(4):79–82, July-August 2003.
13. 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.
14. Gnutella Protocol Specification v0.6. http://rfc-gnutella.sourceforge.net.
15. P. Grace, G. Coulson, G. Blair, L. Mathy, W.K. 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 Applications (DOA04)*, pages 1463–1481, Cyprus, October 2004.
16. M. Hiltunen and R. Schlichting. A Configurable Membership Service. *IEEE Transactions on Computers*, 47(5):573–586, 1998.
17. D. Hughes, I. Warren, and G. Coulson. AGnuS: The Altruistic Gnutella Server. In *Proceedings of the 3rd International Conference on Peer-to-Peer Computing (P2P2003)*, pages 202–203, Linkoping, Sweden, September 2003.
18. N. Hutchinson and L. Peterson. The x-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
19. F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the 2nd ACM/IFIP International Conference on Middleware*, pages 121–143, New York, NY, USA, April 2000.
20. B. Li, J. Guo, and M. Wang. iOverlays: A Lightweight Middleware Infrastructure for Overlay Application Implementations. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 135–154, Toronto, Canada, November 2004.
21. 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.
22. H. Miranda and L. Rodrigues. Communication Support for Multiple QoS Requirements. In *Proceedings of the 3rd European Research Seminar on Advances in Distributed Systems (ERSADS99)*, Madeira Island, Portugal, April 1999.
23. ISO Reference Model for Open Distributed Processing. http://www.dstc.edu.au/Research/Projects/ODP/standards.html.
24. N. Parlavantzas, G. Coulson, and G. Blair. An Extensible Binding Framework for Component-Based Middleware. In *Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)*, pages 252–263, Brisbane, Australia, September 2003.
25. 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.
26. M. Roman, F. Kon, and R. Campbell. Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online*, 2(5), August 2001.

27. 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.
28. I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001 Conference*, pages 149–160, San Diego, CA, USA, August 2001.
29. 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.
30. R. van Renesse, K. Birman, and S. Maffeis. Horus, a Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.
31. 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.
32. P. Wojciechowski, O. Rutti, and A. Schiper. SAMOA: A Framework for a Synchronisation-Augmented Microprotocol Approach. In *Proceedings of the 18th IEEE Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, April 2004.