A Reflective Middleware to Support Peer-to-Peer Overlay Adaptation

[1]Gareth Tyson, [1]Paul Grace, [1]Andreas Mauthe, [1]Gordon Blair and [2]Sebastian Kaune

[1]ComputingDepartment, InfoLab21, Lancaster University, Lancaster, UK
[2]KOM Multimedia Communications, Technishe Universität Darmstadt, Germany
[1]{g.tyson, p.grace, andreas, gordon}@comp.lancs.ac.uk, [2]kaune@kom.tu-darmstadt.de

Abstract. As peer-to-peer systems are evolving from simplistic application specific overlays to middleware platforms hosting a range of potential applications it has become evident that increasingly configurable approaches are required to ensure appropriate overlay support is provided for divergent applications. This is exacerbated by the increasing heterogeneity of networked devices expected to host the overlay. Traditional adaptation approaches rely on simplistic design-time isolated fine-tuning of overlay operations. This, however, cannot fully support the level of configurability required by next generation peer-to-peer systems. To remedy this, a middleware overlay framework is designed that promotes the use of architectural reconfiguration for adaptive purposes. Underpinning this is a generic reusable component pattern that utilises software reflection to enable rich and extensible adaptation of overlays beneath divergent applications operating in heterogeneous environments. This is evaluated through a number of case-study experiments showing how overlays developed using the framework have been adapted to address a range of application and environmental variations.

Keywords: Adaptation, peer-to-peer, reflective middleware

1 Introduction

As distributed computing has moved towards increasingly decentralised models it has become evident that responsive and extensible adaptation mechanisms are integral for real-world deployment. Peer-to-peer networking is a prominent example of such a technology. By pushing functionality to the edge of the network it is possible to utilise the extensive resources available at end-hosts. However, by doing so it means that it is necessary to execute system functionality in increasingly uncontrolled and diverse environments, ranging from stable and well-connected desktop computers to low-power embedded devices.

This increasing diversity raises the question of how a system can be expected to effectively adapt in continually evolving operating environments. Early peer-to-peer systems were strongly bound to their application, however, recently there has been a move towards utilising peer-to-peer overlays as a middleware platform for a variety

of applications to operate over e.g. [12][13]. This means that overlays now must not only adapt to their environment but also to the requirements of any applications built over them. Whereas traditional mechanisms (e.g. parametric adaptation) have proved adequate in earlier peer-to-peer overlays that are restricted to a single application, it is clear that they are severely limited in their scope (e.g. file sharing). This becomes evident when deploying such overlays in diverse operating environments below various applications e.g. performing video streaming in MANETs.

This paper designs and evaluates a middleware overlay framework that promotes and utilises extensible architectural adaptation. Central to this design is the use of *abstraction*, *reconfigurable component-based engineering* and *reflection* to facilitate the convenient and extensible adaptation of node behaviour in both local and distributed settings. The rest of the paper is structured as follows. Section 2 offers a background to the work. Section 3 then describes the overlay middleware framework alongside the key principles of its operation. Subsequently in Section 4, a component pattern is described to show how overlays can be developed in the framework. Section 5 evaluates the overlay framework primarily using case-study experiments to highlight the capabilities of architectural adaptation. Lastly, Section 6 concludes the paper outlining areas of future work.

2 Background and Motivation

There has been a large body of work carried out into peer-to-peer networking. This has focussed on the construction of increasingly sophisticated and novel designs addressing application areas such as video streaming [25], distributed searching [6] and distributed object location [22]. These systems have generally been developed using 'ad-hoc' overlay-specific approaches to adaptation, focussing on adaptation for maintenance purposes [17][21][22] as well as optimisation purposes [1][2][4].

These adaptation approaches can be separated into two primary groups (although other categorisations also exist e.g. [14]). The first category we term *parametric adaptation*. This is discrete parameter adaptation based on variable inputs to a fixed algorithm. For instance, GIA [6] utilises parametric adaptation when constructing its topology. This is done by selecting different neighbourhood sizes based on node capabilities. The second category we term *policy adaptation*. This is performed using variable sets of algorithms that are exchanged during runtime. For instance, BitTorrent [2] uses policy adaptation by utilising different chunk selection algorithms based on the current download status.

Whereas these mechanisms have proved adequate in traditional peer-to-peer systems (e.g. file sharing) it is evident that their potential is limited in next generation applications and networks. This is because both mechanisms require the design-time isolation and implementation of adaptive functionality. This limits flexibility when deployed in diverse environments possessing unpredictable characteristics. For instance, an unstructured search overlay might adapt its resilience algorithms by parametrically altering the number of neighbours it utilises. This, whilst adequate in a traditional deployment, does not sufficiently support adaptation if ported to a mobile ad-hoc network (MANET). This is because it would also be necessary to adapt a

number of alternate concerns. A new localised neighbour selection policy would be required to limit egress communications. Similarly the forwarding algorithms would require adaptation to exploit the broadcast nature of the MANET environment. Further, the maintenance procedures would require modification to respond to the high latency, transient nature of peers.

These criticisms were first provided by Oreizy et al [19]. This seminal paper promoted the use of well-defined software architectures for supporting system evolution. Later work such as [3][12] further identified the advantages of software architectures in adaptive system design. This work dictates that entire systems are built from well-defined independent software entities called *components* [8] that possess fixed capabilities (interfaces) alongside well-defined dependencies (receptacles). Through this, specialised and adaptive systems can be built by dynamically interconnecting optimal component interfaces and receptacles during runtime. The complete construction of systems from abstracted components further opens up adaptation to any aspect of the system rather than limited sets of functionality identified at design time.

In the last decade a number of adaptive systems have been developed using these principles. To support these, a number of lightweight *component models* have emerged e.g. Fractal [5] and OpenCOM [8]. These component models support adaptation by managing such things as component dependencies and runtime reconfiguration. Notably these two examples further support the concept of *reflection* [16]. This is the ability for a system to gain introspection to the capabilities and behaviour of its constituent components. This allows a system to match its requirements to its available components to build new and extensible *configurations*.

A range of reflective middleware has been developed for such things as QoS [7], remote method invocation [3] and sensor networking [12]. However, only limited work has been carried out into the architectural adaptation of overlays. The Open Overlays project [12] carried out initial work in the area, however, this takes a coarse grained approach. RaDP2P [15] and AdaPtP [14] look at standardised adaptation for peer-to-peer systems. These, however, focus on adaptation strategies rather than the underlying platform for adaptation i.e. parametric, architectural etc. PROST [20] utilises abstraction for overlay adaptation (specifically for structured overlays). It does this by using a standardised API (e.g. Dabek et al [9]) and using different overlay networks behind it. This, however, only offers very simplistic adaptation strategies that do not offer sufficiently fine-grained, rich adaptation for real-world usage.

3 Principles behind Reflective Overlay Adaptation

This section introduces the principles behind architectural reconfiguration and how they can be exploited by peer-to-peer system to achieve flexible adaptation.

3.1 Abstraction and (Re)Configurable Software Design

Abstraction is an important concept in software engineering. It forms a platform for both software evolution and system adaptation. It involves the modularization of

system functionality into well-defined abstractions, shown as small boxes in Fig 3.1; these each represent one abstracted aspects of the overlay's functionality. This can be done in a very coarse sense (e.g. the overlay as one unit) or alternatively in a very fine grained sense (e.g. placing every method behind an independent abstraction). By taking a finer-grained approach the entire system is opened up to adaptation by supporting the modification of any system aspect behind its interface. This therefore allows multiple implementations of the same interface to be dynamically exchanged on a node to react to context variations in the operating environment; these individual implementations are termed *pluggable components*.

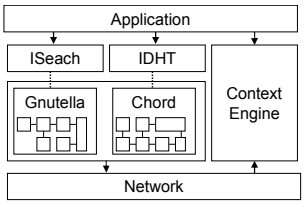

Fig 3.1 Overview of Framework Configured with ISearch and IDHT

An important design concern is the way that system functionality is separated into these independent components. A standardised approach to functional separation is termed a *component pattern*. The framework's default component pattern for implementing overlays is described in Section 4. As well as these internal abstractions, however, the framework also exploits an external abstraction. This is the provision of standardised access to the overlay allowing the application to seamlessly operate with the overlay, even during adaptation. This can be seen in Fig 3.1 with the ISearch and IDHT abstractions being offered to the application. These abstractions, in turn, are mapped to the underlying functionality of supporting overlays (in this example, Gnutella [6] and Chord [22]). This allows alternative overlays (or adapted configurations of the same overlay) to be interchanged behind the abstractions e.g. exchanging Chord for KAD [18] in unreliable environments. A number of other abstractions are also currently available in the framework, including: multicast, group messaging and stored distribution. Lower level native abstractions are also provided to support access to the base overlay operations i.e. routing messages.

3.2 Reflection and Context-Aware Configuration

To perform architectural adaptation it is necessary for a node to be aware of its own software structure. Reflection is the enabling technology behind this; it allows a piece of software to inspect and manipulate its own implementation. This allows the node to explicitly state which components are operating and how they interact. Importantly, it further allows the node to dynamically replace such components to best serve the host. To enable this selection process, components are associated with meta-tags describing particular attributes of their behaviour in name-value pairs.

Composites of components are further constructed to build more sophisticated bodies of functionality (for example, in Section 4 it is described how a control component is composed of finer-grained components). These composites are represented by *configuration scripts*. Simple scripts can dictate the interconnection of

two components whilst more sophisticated scripts can describe the construction of a fully functioning node. Importantly, each script is required to offer meta-information that describes its characteristics in the same way that components are.

When an application is initiated it must provide the middleware with two sets of information. This first set is its *functional requirements*; these are all the application's overlay interface requirements (e.g. IDHT, ISearch etc). At runtime, this information is passed to the middleware's context engine. This is a decision engine that selects the optimal configuration for a particular set of requirements. It iterates through all available components and scripts to locate ones that offer the desired interfaces.

Once a set of compatible configurations have been selected, the context engine inspects the next set of requirements provided by the application: the *behavioural requirements*. These are rules that define the preferred meta-values of the overlay's constituent components. For instance, a behavioural requirement could be that a forwarding algorithm must be able to route in log(N) time. This would indicate that the IRouting component attached to the interface must achieve log(N) efficiency. This would be represented through the rule:

[Time_complexity=='log(N)']

Due to the distributed nature of peer-to-peer systems it is often necessary to provide coordinated reconfiguration between multiple peers. To support this, the middleware utilises a *just-in-time* approach that exploits reflection to allow nodes to dynamically adapt as and when other peers require them to. One reflective attribute provided by components and configuration scripts is the protocol messages that they can process. If a message is received at the transport layer of the framework that the current configuration cannot handle then it is reconfigured to process the message. This is done by constructing a new behavioural requirement that includes the ability to handle the unknown protocol message identifier e.g.

[Protocol_support=='LB_Routing:LoadUpdate']

This rule is then passed to the context engine which reconfigures the node based on the new requirements. This results in communities of dynamically adapting peers cooperating within the same configuration. By utilising the middleware's fine-grain component pattern (described in Section 4) this can be performed in a low-overhead fashion through the adaptation of small aspects of functionality.

3.3 Adaptation Policies

The ability to define behavioural requirements allows convenient specialisation of node behaviour. However, it is also beneficial to define explicit situations in which a particular adaptation should take place. The reflective nature of the design means that adaption policies can be externally defined rather than within the overlay code. Developers (or third parties) therefore provide adaptation rule-sets which dictate that a particular configuration should be executed if a node enters a certain state. To do this during runtime, the context engine correlates environmental measurements from various context sources e.g. bandwidth monitors, processor monitors, user behaviour profilers. This information is then compared against the adaptation rules; if a rule is triggered it can then dictate a certain configuration script is executed. This will be explored in more detail in Section 5.

4 Overlay Component Pattern

The previous section has introduced the middleware and its general mechanisms. However, as well as this, suitable software patterns are also required to optimally build overlays. This section present a generic, reusable component pattern in which overlays can be effectively developed for the purposes of adaptation. Fig 4.1 provides an overview using Pastry [21] as an example. It is also important to state that the framework can operate with any component pattern. Details of alternate patterns can be found in our past work [12][[23][24].

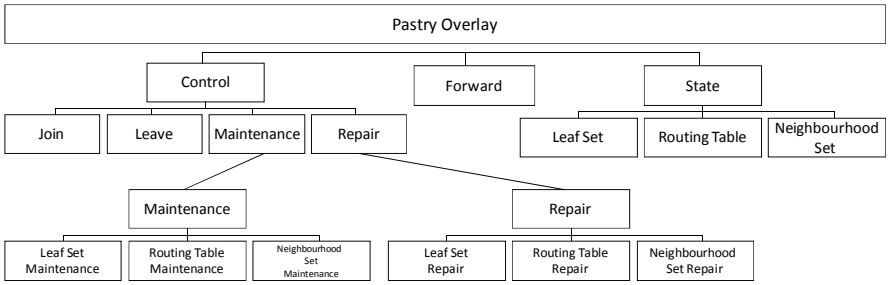

Fig 4.1 A Pastry Implementation in the Framework's Default Component Pattern

4.1 Control

The Control aspects of the pattern deal with managing the overlay. There are four composite components in this branch of the tree: Join, Leave, Maintenance and Repair. These are generic concepts present in all overlays; alongside DHTs these Control aspects have been implemented in a number of other overlays including unstructured search [6], gossip communications [11] and overlay multicast [17].

The *Join* component deals with the joining procedure for a node. In terms of Pastry this refers to locating its position in the topology, collecting the necessary leaf set and routing table members then informing all interested nodes of its arrival. It is important to modularise this concern as the joining procedure largely dictates the structure of the network. Therefore by allowing independent access to the joining process, a developer can conveniently modify the topology of the overlay. For example, the ring topology of the Pastry network can be easily configured to also create links between logically distant peers in order to improve reliability, load balancing or routing.

The *Leave* component handles the removal of a node from the network. This is an effective modularisation as it allows peers in different environments, with different higher level applications to easily perform separate leaving procedures e.g. silent, elegant and daemon leaves.

The *Maintenance* component manages the monitoring of the overlay for changes. This can be for optimisation or alternatively just to ensure the overlay's integrity. There must be an individual maintenance component for each State component in the system. This maintenance component therefore monitors the necessary factors that are essential for its particular state's integrity. A wide range of possible reconfigurations

can occur involving the maintenance procedures. For example, one node might use low overhead keep-alive messages whilst another would employ secure and resilient procedures involving frequent state broadcasts, certificate exchange and encryption.

The *Repair* component deals with repairing (or optimising) any problems located by related Maintenance components. Therefore, if the Leaf Set Maintenance component discovers the loss of a leaf set member then the Leaf Set Repair component is required to correct the issue. Similarly, if the Routing Table Maintenance component locates a superior routing entry then the Routing Table Repair component is responsible for implementing the state changes. In a similar vein to the Maintenance component, it is necessary for individual Repair components to be developed for each State component. This allows triplets of components (Maintenance, Repair and State) to be reused together. Further, it allows much finer-grained adaptation to take place without having to involve the adaptation of multiple state sets at the same time.

4.2 Forward

The Forward component deals with routing in the overlay. This can be simplistic as in the case of Gnutella [6] and CoolStreaming [25] or alternatively quite complex as in the case of Pastry [21]. Unlike the Control aspects, Forward is not separated into sub-components but is left as a single component. This is due to the well-defined and simple nature of forwarding algorithms. This can be contrasted with Control aspects which contain a wide range of diverse functionality. The Forward component is a very important modularization; this is because its reconfiguration allows rich variations in behaviour. For example, our Gnutella implementation can conveniently adapt to utilise gossip-based, random walk or semantic searching.

4.3 State

The State components embody the data structures required by each node in the overlay. Each data structure is embodied in its own component to improve the reusability and portability of such entities. It thus becomes possible to permanently associate the State components with their respective Control components. This therefore allows reconfiguration to occur without the need to move state data between new and old components.

5 Evaluative Case Studies

The framework has been implemented in Java using the OpenCOM (v1.4) component model [8]. It is part of a larger architecture called Juno [24]; this is a configurable middleware designed to address the heterogeneity of next-generation content distribution. It does this by underpinning services and delivery mechanisms with the overlay framework. A number of overlays have been implemented using this component-based approach. These include Chord [22], SCAMP [11], BitTorrent [2],

TBCP [17] and Pastry [21]. Using these implementations a number of case-studies are described to highlight the capabilities and limitations of the framework. An overhead study is also provided for completeness. This evaluation, however, does not provide a quantitative study of individual overlay implementations. This is because such a study would evaluate the performance of an individual overlay or algorithm. Instead, we show how a number of systems can perform adaptation through the reflective, architectural reconfiguration of the framework. Performance details of the individual adaptation algorithm are provided in the references.

5.1 Case-Study Experiments

5.1.1 Experiment A: Local and Community Adaptation

This experiment investigates the community based adaptation of peers; first in a local sense then in a distributed one. Specifically, it looks at adapting a Pastry node to distribute load balancing information amongst routing neighbours. This adaptive mechanism (outlined in [1]) has been designed to alleviate the load on certain areas of the network. It further enables peers with low resources to contribute less, therefore improving routing performance. When a node reaches a certain load it begins to attach load tags to any sent messages. These are then read by downstream routing neighbours that, in turn, show preference to less-loaded routing choices. Further, the maintenance algorithms adapt to propagate live load information about the node. This allows nearby routing peers to maintain an accurate, real-time view of the vicinity's routing loads. This rich adaptation cannot be natively supported by a conventional Pastry implementation without redevelopment.

The framework implements adaptation through externalised adaptation rule-sets. The following rule is added to the rule-set to define the load balancing adaptation:

if [load > max_load] **do** config *load_balance*
else do config *standard_pastry*

This indicates that a node should execute the load_balance script if its current load exceeds the maximum load. Similarly, if this load decreases it should initiate the standard_pastry script. To do this, the rule-set configuration file is therefore required to define the max_load threshold alongside the calculation of the load variable.

Due to the nature of the adaptation it is identifiable that two aspects are involved: *maintenance* and *forwarding*. The load_balancing script therefore dictates the replacement of the existing maintenance and forwarding components with their load balancing equivalents. By separating these aspects as independent pluggable components it is therefore possible to dynamically alter their behaviour by replacing them in the architecture. This simplifies the adaptation process by building well-defined algorithms embodied in components that are open to third-party coordination through reflection, configuration and later development.

During the reconfiguration process the peer is placed in a quiescent state. The framework achieves this by completing all component interactions before buffering all future interactions. Similarly, remote interactions are queued. Once the reconfiguration is complete, the peer is reactivated. At this point, the software architecture has the new LB_Maintenance and LB_Forward components attached.

Due to abstracted component interaction, it is possible to continue the node's operation without changes to other existing components in the architecture.

Once the new components begin execution it is necessary to utilise protocol messages that are not natively understood by other load balancing unaware peers. To overcome this, the framework exploits *just-in-time* distributed adaptation. When an unknown protocol message is received by the remote host, the node's context engine inspects the protocol handling capabilities of all its available components and configuration scripts to locate configurations capable of understanding the message. In this scenario this obviously results in the execution of the *load_balance* script which dictates the installation of the LB_Forward and LB_Maintenance components. Once the two components have been installed, the Transport layer resumes execution from the initial receipt of the load balancing message. This results in the message being passed to the LB_Forward component and being correctly processed.

This lightweight process (c.f. Section 5.2) is carried out for every neighbour that receives the load balancing information. This allows the peers to perform coordinated adaptation in small communities; further, by dynamically acquiring new components it becomes possible to extend adaptation mechanisms at run-time.

This experiment highlights the framework's ability to dynamically install rich new adaptive functionality on a node. Particularly, this shows how the (re)configurable approach can extend the adaptive model beyond that of 'fine-tuning' by allowing entire bodies of functionality to be dynamically modified behind abstracted interfaces. Importantly this process has shown to be possible in a distributed sense. Conventional adaptive approaches are limited to well-defined strategies that are ubiquitous to all member nodes at design-time. The experiment shows, however, that it is possible to deploy adaptive strategies dynamically between subsets of peers at run-time.

5.1.2 Experiment B: System-Wide Adaptation

The previous experiment has shown how individual nodes can adapt in communities to improve performance. This experiment shows how system-wide adaptation strategies can further improve performance. This indicates that all peers operate the same adaptation to create system-wide behaviour. Specifically, in this experiment the Pastry overlay is separated into two groups of peers based on their capabilities. Reliable peers perform the traditional role of Pastry peers and form the *routing set,* whilst unreliable peers that do not contribute to the routing procedure form the *client set*. This separation significantly improves resilience and performance by removing transient peers from the system [4].

A new peer joining the system is initially configured as a client node by executing the pastry_client script. Once a burn-in time has been reached, the peer then adapts itself by installing the necessary routing components. A node is reconfigured to operate as a client again if its environment becomes unreliable. This is represented in the adaptation rule-set using the rules:

if [online_time > burn_in && isReliable()==true]
    do config *pastry_router*
else do config *pastry_client*

The rule set contains the method isReliable(). This allows the rule-set to ascertain whether the peer is currently considered reliable. This script can either utilise default framework implementations or alternatively a specialised overlay implementation.

The pastry_client script contains instructions to install two components: Client_Join and Client_Forward. It is evident that the pastry_client script does not need to instantiate all the components dictated in the software pattern outlined in Section 4. This is because the simplicity of the client peers reduces the actual number of required components. Through the provision of a fine-grained architecture it therefore becomes easy to use subsets of functionality required for particular configurations. The Client_Join component simply creates a point-to-point connection with a node from the routing set whilst the Client_Forward component, in turn, forwards all messages through this proxy.

Once a peer is considered eligible for membership of the routing set, it is reconfigured by executing the pastry_router script. This involves detaching the Client_Join and Client_Forward components and attaching a full set of Pastry components. Once this reconfiguration process has completed the Join component executes the standard joining algorithm.

This case-study has shown the framework's ability to support system-wide adaptive strategies beyond the local and neighbourhood scope of the previous experiment. To facilitate this it exploits independent access to the joining and routing mechanisms. This represents an adaptive process that exceeds the capabilities of existing Pastry implementations (e.g. FreePastry [10]). This is because traditional adaptive mechanisms are restricted to adapting aspects that are isolated at design-time. The join mechanism is not utilised in conventional Pastry adaptation and is in general rarely isolated for adaptation. Therefore, by constructing nodes from open component patterns that provide access to all functionality, it becomes possible to implement adaptive strategies during run-time that exceed original design plans.

5.1.3 Experiment C: Application-Driven Adaptation

The previous experiments have shown how environmental factors can drive the adaptive configuration of overlay behaviour. This experiment investigates the adaptation of the middleware when operating beneath divergent applications. We define a divergent application as one that has a range of distributed interaction requirements. Specifically, this experiment looks at an application that offers stored and streamed video delivery alongside a search facility. Such a system requires at least three types of overlay, creating significant complexities for developers.

When the application is initialised over the middleware it provides a list of its required abstractions (i.e. ISearch, IStreaming and IStoredDelivery); these are termed *functional requirements*. The middleware's context engine then locates all the overlay configuration scripts that offer at least one of these interfaces. Once this has been done it is important to differentiate between overlays offering the same interface but possessing different characteristics. This is achieved through the construction of *behavioural requirements*. To investigate this, three different uses of the video application are investigated; firstly, the use of the application for distributing lectures on a campus; the second is distributing movies in an Internet-scale situation; and the third is distributing corporate videos amongst a number of offices.

This case-study obviously requires that different search mechanisms are utilised when the application is operating in different environments. When the application is bootstrapped, it defines its behavioural requirements by describing the required characteristics of any overlay operating behind the ISearch abstraction e.g.

Abstraction::overlays.interfaces.ISearch

1: [Size > 1000]

2: [Multi_Keyword==true]

3: [Fuzzy==false]

This example indicates that the ISearch overlay configuration selected must be able to support a user group greater than 1000 and support multiple keyword searching whilst not requiring fuzzy matching. Any attributes can be attached to components/scripts allowing highly extensible application driven configuration to take place. This process therefore encourages the use of a domain specific ontology.

Table 5.1 shows the behavioural requirements of the three scenarios. The lecture scenario instantiates a Gnutella [6] overlay network due to its small size; the movie scenario instantiates a server based search mechanism as it supports large-scale fuzzy searching; lastly, the corporate scenario utilises Pastry [21] as it requires up to 50,000 users but without fuzzy search support.

	Size	**Keyword**	**Fuzzy**	**Overlay**
Lectures	<2000	TRUE	TRUE	Gnutella
Movies	>100,000	TRUE	TRUE	Server
Corporate	100 – 50000	TRUE	FALSE	Pastry

Table 5.1 Behavioural Requirements of ISearch and Selected Overlay

This process can also be performed dynamically; for instance, if the user decides to cease watching the lecture service and switch to the movie service, the application will submit new behavioural requirements to the context engine. The context engine then locates a more appropriate search mechanism using the meta-values provided by the different overlay configuration scripts available. This process yields the *Server* script which results in the Gnutella functionality being shutdown and detached from the application and replaced with the Server search mechanism. In future, whenever the application utilises the ISearch abstraction it will therefore be actually utilising a client-server search rather than Gnutella. This is performed transparently, however, to ensure consistency the application can stop any undesired reconfigurations e.g. if certain search information is not replicated on the multiple networks.

This experiment has highlighted how architectural adaptation can support applications with non-fixed behavioural requirements. Specifically, it has been shown that an application can be conveniently deployed for use in a number of different application scenarios without intensive application coding. Instead, through abstracted reconfiguration the context engine can adapt the overlay (and therefore the application) without modifications to application code.

5.2 Resource Overhead

Due to the necessity to manipulate software components during the adaptation process, an added overhead is introduced. This section briefly discusses the resource overheads involved. All tests are performed on a 2.1GHz Intel Core 2 Duo processor; 4 GB RAM; Sun JVM 1.6.0.5.

5.2.1 Processing Overhead

We first measure the processing costs of utilising the framework; Table 5.2 shows the maximum processing throughput of overlays built in the framework. This has been measured by benchmarking the maximum number of component invocations possible. For comparability, the same process is carried out with Java interfaces.

It can be seen that the framework has a noticeable decrease in throughput. This creates a trade-off between flexibility and overhead with the framework sacrificing processing capacity to support runtime abstraction and reconfiguration. It should be noted, however, that the ability to utilise lightweight configurations (e.g. Experiment B) can improve the overlay processing overhead for low-capacity peers.

Type	(Invocations/Second)
Native Java	15.863570×10^6 (16 million)
Framework	3.222367×10^6 (3 million)

Table 5.2 Throughput of Component Interactions

5.2.2 Memory Overhead

To validate that the framework does not create unacceptable memory consumption, Table 5.3 shows the dynamic footprints of a subset of the framework's overlay implementations; these include the JVM. These implementations have been developed using a range of component patterns and offer different levels of complexity. They all, however, show acceptable memory footprints when compared to existing Java implementations (e.g. FreePastry's [10] footprint is 12,232KB).

	Pastry [21]	Chord [22]	SCAMP [11]	TBCP [17]
Footprint (KB)	9,656	11,932	13,708	15,144

Table 5.3 Memory Footprint of Overlays

5.2.3 Reconfiguration Time

To show the cost of architectural reconfiguration, Table 5.4 shows the time taken to execute reconfiguration in the middleware. Experiments A is shown as it offers very fine-grained adaptation whilst Experiment B involves a much heavier reconfiguration. Importantly, the application logic has been removed to ensure that the measurements aren't affected by overlay specific concerns.

	Bootstrap (µSecs)	Adaptation (µSecs)
Experiment A	12831	592
Experiment B	284	1617

Table 5.4 Reconfiguration Time

Experiment A has a high configuration time during bootstrapping. This is because it has to install the full overlay. This can be contrasted with its adaptation process that requires only 5% of the time. This is because the framework's fine-grain component pattern allows the adaptation to take place by replacing only two small components. In contrast, Experiment B has a low configuration time during bootstrapping. This is

because it exploits the lightweight pastry_client configuration. Its adaptation process, however, it significantly more complex as it requires the installation of the routing aspects of the Pastry overlay. Importantly, the adaptation process is shown not to require extended periods of reconfiguration.

6 Conclusion and Future Work

This paper has investigated the potential of exploiting architectural reconfiguration for the adaptation of peer-to-peer systems. To this end, an overlay middleware has been designed that exploits the reflective component-based implementation of overlays. Through generic, reusable patterns it becomes possible to host applications over specially configured overlays dynamically selected at runtime. This enables broad functional requirements to be satisfied by the framework through the instantiation of multiple concurrent overlays behind various standardised abstractions. An important feature of this procedure is the framework's ability to consider behavioural requirements during the selection of effective overlay configurations. This allows the application to transparently operate over the most effective overlay for its (dynamically changing) requirements. To supplement this, the framework also supports the explicit scripting of adaptive algorithms for any componentised aspect of the system, allowing externalised policies to be dynamically added during runtime.

A number of interesting areas of future work exist. We consider it important to establish such an engineering approach to overlay development. Therefore further investigation into generic patterns that can match the requirements of diverse overlays is necessary. This should not be restricted to adaptive considerations but also provide support for non-functional concerns such as QoS, fault-tolerance, resilience and error management. Also, large-scale investigations must be performed to observe the effects that local and community adaptation has on the system as a whole. Alongside this, important areas also include security and runtime configuration checking.

Acknowledgements

This work is supported by the European Network of Excellence CONTENT (FP6-IST-038423)

References

1. Bianchi, S., Serbu, S., Felber, P., and Kropf, P. Adaptive Load Balancing for DHT Lookups. In Proc. Intl Conference on Computer Communications and Networks, Arlington, Virginia (2006).
2. BitTorrent Specification. *http://www.bittorrent.org/beps/bep_0003.html*.
3. Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N. and Saikoski, K., The Design and Implementation of Open ORB V2. In IEEE Distributed Systems Online (2001).
4. Brampton, A., MacQuire, A., Rai, I. A., Race, N. J., and Mathy, L. Stealth Distributed Hash Table: A Robust and Flexible Super-Peered DHT. In Proc. ACM CoNext, Lisbon, Portugal (2006).

5. Bruneton, E., Coupaye, T., Leclerc, M., Quema, V. and Stefani, J-B. An Open Component Model and its Support in Java. In Proc. Intl. Symposium on Component-Based Software Engineering Edinburgh, Scotland (2004).
6. Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., and Shenker, S. Making Gnutella-like P2P Systems Scalable. In Proc. SIGCOMM, Germany (2003).
7. Coulson, G. A Configurable Multimedia Middleware Platform, IEEE Multimedia Magazine, vol 6, issue 1, pp 62-76, IEEE Press, January-March (1999).
8. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, Jo and Sivaharan, T. A Generic Component Model for Building Systems Software. In ACM Transactions on Computer Systems, 27(1):1-42, February (2008).
9. Dabek, F., Zhao, B., Druschel, P., Stoica, I. Towards a common API for structured peer-to-peer overlays. In Proc. IPTPS Berkeley, CA, (2003).
10. FreePastry. Available at http://freepastry.org/
11. Ganesh, A., Kermarrec, A. and Massoulie, L. SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In Proc 3rd Intl. Workshop on Networked Group Communication, London, UK. (2001).
12. Grace, P, Coulson, G., Blair, G., Mathy, L., Yeung, W., Cai, W., Duce, D., and Cooper, C. GridKit: Pluggable Overlay Networks for Grid Computing. In Proc. Intl. Symposium on Distributed Objects and Applications, Larnaca, Cyprus (2004).
13. Gu, X., Nahrstedt, K., and Yu, B. SpiderNet: An Integrated Peer-to-Peer Service Composition Framework. In Proc. 13th IEEE International Symposium on High Performance Distributed Computing, Honolulu, HA (2004).
14. Hughes, D. AdaPtP - a Framework for Building Adaptable Peer-to-Peer Systems. PhD Thesis, Lancaster University (2007).
15. Hughes D., Coulson, G., and Warren, I. A Framework for Developing Reflective and Dynamic Peer-to-Peer Networks (RaDP2P). In Proc. 4th IEEE International Conference on Peer-to-Peer Computing, Zurich, Switzerland (2004).
16. Kon, F., Costa, F., Blair, G., and Campbell, R. H. The Case for Reflective Middleware. Commun. ACM 45, 6 (Jun. 2002), 33-38.
17. Mathy, L., Canonico, R. and Hutchinson, D. An Overlay Tree Building Control Protocol. In Proc. Intl. Workshop on Group Communications, London, UK (2001).
18. Maymounkov, P. and Mazières, D. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In Proc. IPTPS, Sweden (2002).
19. Oreizy, P., Medvidovic, N., and Taylor, R. N. Architecture-based runtime software evolution. In Proc. Intl. Conference on Software Engineering Kyoto, Japan, (1998).
20. Portmann, M., Ardon, S., Senac, P., and Seneviratne, PROST: A Programmable Structured Peer-to-Peer Overlay Network. In Proc. Intl. Conference on Peer-To-Peer Computing, Zurich, Switzerland (2004).
21. Rowstron, A., Druschel, P., Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In Proc. Middleware, Heidelberg, Germany (2001).
22. Stoica, I., Morris, R., Karger, R.D., Kaashoek, M., Balakarishnan, H. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In Proc. of ACM SIGCOMM, San Diego, (2001).
23. Tyson, G. Component Based Overlay Development in Gridkit. Available at *http://www.comp.lancs.ac.uk/~tysong/*. MSc Thesis, Lancaster University.
24. Tyson, G., Mauthe, A., Plagemann, T. and El-khatib, Y. Juno: Reconfigurable Middleware for Heterogeneous Content Networking. In Proc. 5th Intl. Workshop on Next Generation Networking Middleware (NGNM), Samos Islands, Greece (2008).
25. Zhang,.X, Liu, J., Li, B., and Yum, T.S.P. CoolStreaming: A Data-driven Overlay Network for Live Media Streaming. In Proc. IEEE Infocom, Miami, FL (2005).