View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Lancaster E-Prints

Fine Grained Component Based Engineering of Adaptive Overlays: Experiences and Perspectives

Gareth Tyson, Paul Grace, Gordon Blair and Andreas Mauthe

Computing Department, InfoLab21, Lancaster University, Lancaster, UK
{g.tyson, p.grace, gordon, andreas}@comp.lancs.ac.uk

Abstract. Recent years have seen significant research being carried out into peer-to-peer (P2P) systems. This work has focused on the styles and applications of P2P computing, from grid computation to content distribution; however, little investigation has been performed into how these systems are built. Component based engineering is an approach that has seen successful deployment in the field of middleware development; functionality is encapsulated in 'building blocks' that can be dynamically plugged together to form complete systems. This allows efficient, flexible and adaptable systems to be built with lower overhead and development complexity. This paper presents an investigation into the potential of using component based engineering in the design and construction of peer-to-peer overlays. It is highlighted that the quality of these properties is dictated by the component architecture used to implement the system. Three reusable decomposition architectures are designed and evaluated using Chord and Pastry case studies. These demonstrate that significant improvements can be made over traditional design approaches resulting in much more reusable, (re)configurable and extensible systems.

Keywords: Peer-to-peer (P2P), overlays, component based engineering, configuration, adaptation, functional evolution.

1 Introduction

Over recent years there has been an explosion in the number of peer-to-peer (P2P) systems under development, addressing a number of issues, ranging from grid computation to content distribution [1][12][23][24]. Whereas much effort has been put into the development of these novel systems, little research has been promoted into how these overlays are built. This has led to a huge array of development approaches being utilised, ranging from the use of standardised APIs [8] to simple monolithic designs.

This non-formalised approach, however, ignores the potential that software engineering principles can bring. One particular approach that has found much success in the field of middleware systems is the use of component based design to facilitate such things as configurability, adaptability and reusability. These systems separate functionality into independent pluggable entities called components. These systems can then be constructed and reconstructed from such components to

specialise performance for different environments. In the field of P2P overlays this involves nodes being constructed from a subset of components from a repository to offer optimal performance for the existing network constraints and requirements. Further to this, the nodes can be dynamically reconfigured to respond to various events in the system by plugging in different components. Alongside these advantages, the use of components also brings greater software engineering benefits, promoting the reuse of components and functional extensibility for easy development, deployment and maintenance of systems.

The effectiveness of a component based system is largely dependent on the way in which functionality is separated into the components. These components then make up an architecture, or *pattern*, in which they are interconnected. This paper presents and analyses three such component patterns aimed at the design of P2P overlays. These patterns are based on the Gridkit Overlay Framework [14] and are designed to assist developers in the rapid development of (re)configurable overlays for deployment in heterogeneous environments. Through this approach we have implemented a number of overlays, including SCAMP [13], SCRIBE [6], PAST [10] and TBCP [18]. To aid in this investigation, however, we focus on the development of two component based overlays, Chord [24] and Pastry [23]. An alternate evaluation of this work can be found in [27], focussing on re-configurability aspects of a variety of different overlays.

We show that, by designing P2P overlays in this fashion, a large number of advantages can be gained. Existing work in has focussed on coarser grained patterns, however, we investigate the potential of exploiting the properties of finer grained approaches. It is found that (re)configuration of node behaviour can be dynamically and effectively carried out in a much more elegant and extensible manner compared to more conventional parametric adaptation or coarser grained alternatives. Further to this, design complexity and software engineering aspects are also investigated to show the benefits for software developers.

This paper is presented as follows; Section 2 offers a background overview of the area. Section 3 gives a description of the proposed component architecture. Section 4 then provides a short overview of the evaluative implementation we carried out. Following this, in Section 5, is a detailed overview of (re)configuration in the architecture. Section 6 then provides an evaluation of the non-functional, performance and engineering, properties of the approach. Finally, Section 7 provides a conclusion and shows a number of future areas of work that could be carried out in the field.

2 Related Work

There has been a large body of work carried out in the area of P2P networking. This technology involves utilising the resources of end-hosts to provide a service. One example of such a service is distributed object lookup in which nodes self-arrange to allow them to build a distributed hash table. Examples of these systems are Pastry [23] and Chord [24]; they both share similar facets in that they both build a ring topology. However, whilst Chord routes messages over the ring, Pastry also builds a Plaxton [20] routing tree to pass messages through.

One frequently cited issue with developing such P2P overlays is the extensive coding effort that must be taken to implement a new system. To assist in this, a number of approaches to ease the development costs have been proposed. MACEDON [22], OverML [2] and P2 [16] are high level definition languages that allow developers to define the workings of their overlays without the intensive coding process of dealing with lower level functionality. P2, for instance, allows Chord to be defined using 47 logical rules which can be compared to the original MIT implementation containing thousands of lines of C++ code. However, these produce fixed implementations that cannot be adapted once generated and deployed.

There are also middlewares and application toolkits that provide principled support for P2P application development. JXTA (www.jxta.org) is a framework where P2P applications are developed atop a resource search abstraction; this supports grouping and contacting nodes. This abstraction can be implemented using a number of overlay topologies (e.g. Chord or Pastry). However, implementation follows a black-box approach below the abstraction; this restricts configurability in diverse environments, dynamic adaptation and software re-use.

Component-based middleware is an approach that resolves these issues. This sees middleware being constructed from a set of independent pluggable entities called components. A component is described as a self contained body of code that is accessible by a predefined interface [26]. Overall, the benefits of the component approach are as follows: i) it promotes a high level of abstraction in software design, implementation, deployment and management, ii) it fosters third-party software reuse [8], and iii) it facilitates flexible configuration (and, potentially, run-time reconfiguration) of software. Well-known component models include: EJB [25] and Microsoft .NET [19]; however, these are typically heavyweight and application focused.

In response to this, lightweight component models have emerged (e.g Fractal [4], OpenCOM [7], k-Components [9], Koala [29], Pebble [17] and THINK [11]). Notably, the first three also support reflection-based dynamic adaptation. Reflection allows the current component structure and behaviour to be inspected and adapted at runtime. Their lightweight nature allows them to be used for developing system software as well as applications. For example, they are the enabling technology behind reflective middleware, e.g. OpenORB [3], DynamicTAO [15] and RAPIDware [21]. These middlewares can be configured from a subset of potential components allowing them to be specialised and adapted to different scenarios and environments, making them more flexible and extensible. Further, dynamic adaptation of the constituent configurations brings substantial benefits to the system improving performance and efficiency in the face of fluctuating conditions. Reusability is also a further benefit as the use of standardised components allows different systems to exchange components. We believe that the benefits from such component technologies can similarly better support the development of P2P overlay software.

3 Component Patterns for Overlay Decomposition

One of the fundamental issues involved in designing a component based system is how the developer can most effectively separate the system's functionality into components. The most important decision in designing component architectures is the granularity of decomposition; this represents to what extent the functionality of the overlay has been compartmentalised. A coarse architecture may only consist of a few components whereas a fine grained architecture typically uses a much larger number.

The granularity can be defined in two dimensions: namely, width and depth; the *width* refers to the number of identifiable aspects that a system (or component) can be separated into, whist the *depth* refers to how individual aspects of the system are further decomposed. Hence, our software decomposition diagrams follow a tree structure, where each branch of the tree is a component decomposition.

There are a number of pros and cons involved in using such architectures, often making a trade-off between flexibility and complexity. The reasons for using complex architectures are abundant; fine grained component separation allows independent access to a larger number of components in the system which in turn allows independent access to more specific aspects of the overlay. Therefore, finer grained architectures allow much smaller, more specific aspects of the system to be inspected and modified. By possessing access to these individual aspects of the overlay, increasingly significant levels of (re)configurability can be attained. This, however, is not the only tangible gain to be made; as well as this, other software engineering benefits can be gained such as the easy reuse and extension of functionality.

This section presents three component-based patterns for the implementation of overlay networks, based on the Gridkit Framework [14]. These architectures mandate that the implementation of overlays is performed in a specific manner, separating the functionality of the system into a number of independent components.

3.1 Pattern I: Coarse Grained Decomposition

From the highest level, the architecture can be seen to separate functionality into three separate elements as mandated by the Gridkit Framework [14], shown in Figure 3.1. Gridkit is a component based middleware designed to address the heterogeneous design requirements of modern grid applications. To achieve this, it utilises pluggable P2P overlay components allowing a variety of interaction paradigms and services to operate over a variety of overlay networks. These P2P overlay networks are implemented in three independently pluggable components. The first is the Control component which deals with controlling a node's behaviour such as joining it to the network. The second component is the Forwarding mechanism which contains the required algorithms to route information through the overlay. Finally, there is the State component; every other component uses this to store persistent information in, so to facilitate the reconfiguration of the overlay without concern over the state maintained in individual components.

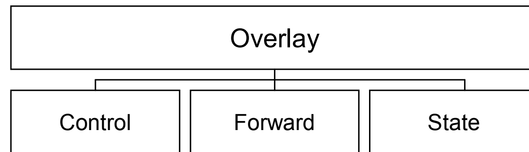

Figure 3.1. Pattern I Overview

When looking at the system from this perspective, it can be seen to suffer from a number of problems. The separation is based on generic, high level definitions of the functionality, deconstructing the overlay into families of algorithms rather than elements that are specifically designed for processes such as reconfiguration or reusability. An example of this is the Control component which encompasses a number of algorithms that manage the overlay ranging from joining procedures to maintaining the network. This approach has been identified as being a suitable methodology for a number of existing overlays such as Chord [24]. However during the implementation of more complex overlays such as Pastry [23] it becomes insufficient. It is therefore necessary to take a closer look at each component to identify the independent aspects that can be extracted and separated out.

3.2 Pattern II: Intermediate Grained Decomposition

A closer look at the architecture outlined previously reveals that the large monolithic elements discussed actually contain a number of individual algorithms. To gain benefits such as configurability, each of these algorithms must be analysed to ascertain the utility of providing independent access to its functionality.

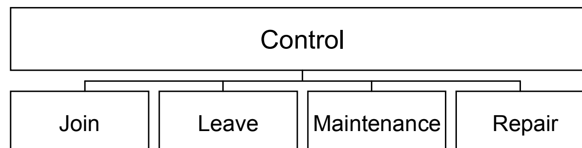

Figure 3.2. Pattern II Control Components

As shown in Figure 3.2, the Control component can be seen to possess a number of individual aspects. The Join component deals with a node joining an overlay; the Leave deals with leaving an overlay; the Maintenance deals with monitoring the status of the overlay whilst, finally, the Repair deals with repairing any problems identified. It can be seen from the outset that these elements represent a substantial amount of the functionality and dictate, to a large extent, performance. For example, the majority of overhead in an overlay will be created by the algorithms embodied in the Maintenance and Repair aspects of the implementation. It is because of this, that these aspects can offer a number of beneficial properties when separated from the rest of the system. This can be clearly seen by looking at heterogeneous environments in which some nodes reside on reliable, wired hosts whilst other run on far more unreliable hosts. In such a scenario it is likely that a superior overlay can be built if each host chose optimal Control components for their environment.

The Forwarding and State components have less identifiable benefits when they are separated. This is because forwarding algorithms are generally uniform in their procedures and comprise of smaller amounts of functionality. For instance, in Plaxton routing [20], to ensure determinism, it is necessary for messages to follow a specific path in the overlay. It is therefore difficult to deconstruct the algorithm further as (re)configuration in this manner could severely compromise the system. Similarly to this, State aspects are limited in areas such as (re)configurability as they are not involved with distributed interactions and behave in a passive manner.

This separation pattern will therefore be effective for overlays which place a high value on basic (re)configurability. The separation of the control elements allow a node to be specialised for individual scenarios. Overlays developed in this manner will not, however, be well suited to reusability as at this granularity most components will still maintain overlay specific functionality that will make it hard to use in a generic way. For instance, the State component will contain all data structures relating to an individual overlay; this will make it inefficient to port to a different system. It is therefore beneficial to inspect an even finer grained approach.

3.3 Pattern III: Fine Grained Decomposition

A number of overlay aspects have been looked at in the previous sections, however it is now necessary to outline an approach to be used that is both generic enough to be used for multiple overlays but specific enough to provide the necessary attributes outlined earlier. Pastry and its data structures have been used to illustrate this pattern; however it is possible for any overlay to be developed in this fashion.

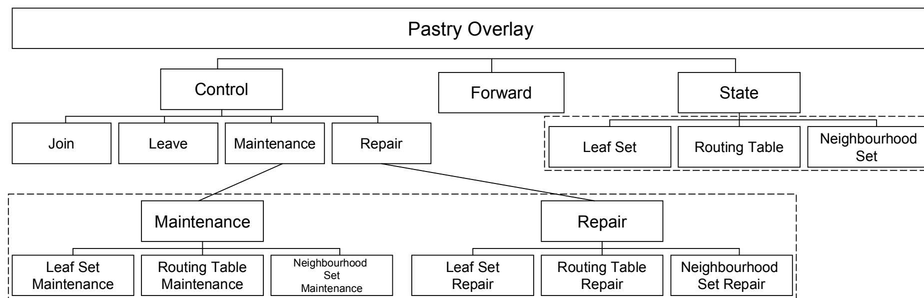

Figure 3.3. Pattern III Overview

Figure 3.3 shows an in-depth view of the proposed component architecture. The further levels of decomposition have been highlighted with dotted lines. The Control element outlined in Pattern II has been separated as suggested into its four constituent elements. Similarly the State component is separated down into a second tier of deconstruction, so to provide for the reusability of its data structures. The Forward component, however, remains as a single unitary element as proposed in Pattern II.

As well as the previously described modules, it can also be seen that the Maintenance and Repair components have been further broken down to embody the various algorithms relevant to maintaining the individual state aspects of the Pastry

design. This improves the reusability of these aspects substantially as reusing entire maintenance or repair components is difficult especially when porting them to different overlays. Each Maintenance and Repair component will now ensure that its respective state table is correct according to some degree of accuracy. This allows the Maintenance and Repair aspects of the system to be reused in accordance with the individual State components; for example, if the Pastry Leaf Set was to be ported to a Chord overlay, it could be done together with the Maintenance and Repair components.

Further to this, the (re)configurability of the system is improved dramatically. This is because, now, the maintenance and repair procedures for each state entity are totally independent. The means that the system can (re)configure these aspects separately without having to interchange both the maintenance and repair algorithms. This is highly beneficial in a number of circumstances as, often, the repair algorithms will remain constant whilst the maintenance elements change. For example, during periods of high node activity, a lazy maintenance algorithm might be selected in which failures are detected passively. Alternatively, if the nodes cease to interact frequently, a probing maintenance algorithm might be employed. Despite these two different approaches, the repair algorithms will remain constant.

Further to this, there are a large number of other potential (re)configurations. For instance, if there is a high turnover in the leaf set an intensive maintenance algorithm might be employed whilst not affecting the routing table maintenance. Alternatively, if misbehaviour is detected in the routing process more secure maintenance and repair algorithms might be installed whilst leaving the leaf set maintenance unmodified.

4 Implementation

To investigate the effectiveness of the component patterns described in Section 3 we have developed a number of overlays, including SCAMP [13], SCRIBE [6], TBCP [18] and PAST [10]. We focus, however, on an implementation of Chord [24] using Pattern I and a Pastry [23] overlay developed using Pattern III. Chord was selected due to its inherent simplicity whilst Pastry was selected due to the complexity of its routing and state elements. This allows a more substantial evaluation and comparison to be performed using a non-component implementation of Pastry as a benchmark.

Both Pastry and Chord were developed using the OpenCOM (v 1.4) component model [7] in Java. Chord's component interactions are performed solely using direct method invocations between the components; each component offers services through public *interfaces* and consumes services through predefined *receptacles*. The decision engine therefore dynamically selects the optimal components and then attaches their receptacles to the appropriate interfaces; this forms *connections* between the components. These connections can then be dynamically modified during runtime.

Pastry, alternatively, utilises an event based interaction system. Using this approach, components generate notifications to inform other components of events that have occurred. Alternatively, events can also be generated to request services from other components. These events traverse the component tree shown in Figure 3.3. It is therefore the decision of each component in the tree how an event is interpreted and

whether they pass it or not. This allows different types of events to be dealt with differently, based on policies implemented in each component. The purpose of this is to allow the effective and extensible addition of functionality to the system without having to reconfigure other aspects of the architecture. Therefore, by decoupling components through events it is only necessary to ensure that all events can be dealt with in the system rather than looking at *how* they are dealt with.

5. Evaluation of Overlay (Re) Configurability

One of the primary aims of utilizing a fine grained model is the ability to (re)configure its behaviour by the architectural modification of the components resident in the system. Configurability refers to a system's ability to be specialised for a particular environment whilst re-configurability refers to its ability to modify itself dynamically to adapt to changes in its environment. Coarse grained approaches are distinctly limited in their (re)configurability as it is only possible to perform architectural modification on each component in the system. Therefore, if there are three components in the system (Control, Forward, State), then it is only possible to configure these three elements independently. Such (re)configurability can be driven by a number of factors consisting of both system requirements and environmental constrains; these can exist in one or more levels:

i) *Network Level* – (Re)Configuration can take place to respond to network variations e.g. bandwidth, packet loss, jitter etc
ii) *Overlay Level* – (Re)Configuration can take place to respond to overlay level variations e.g. malicious peers, routing performance, neighbour selection etc
iii) *Application Level* – (Re)Configuration can take place to respond to application level requirements e.g. data types, interface responsiveness, security etc

To investigate the (re)configurability of the architecture, a maintenance case study is looked at. This highlights how different maintenance and repair algorithms can be utilised based on both application level requirements and environmental limitations. In an un-trusted and unreliable environment (e.g. the Internet), it is beneficial to use rigorous and security conscious algorithms. However, in a closed, trusted, reliable environment (e.g. a campus network), lower overhead algorithms are utilised. This process involves both the Maintenance and Repair components. The Maintenance component implements the different monitoring algorithms whilst the Repair component implements different responses. During bootstrapping, the decision engine selects the optimal components. Run-time variations in the environment and requirements are then responded to accordingly by dynamically interchanging the necessary components.

In the non-component Pastry, sophisticated (re)configuration is not possible. It can only occur in a parametric manner, supporting such things as increasing the size of the leaf set in unreliable environments. The only alternative to this is the process of 'hacking' to modify existing code. This is both time-consuming and inelegant; this, therefore, clearly offers much less flexibility than required to achieve the case study.

The Chord implementation similarly struggles to deal with this type of fine-grained (re)configuration as it is necessary to modify the system on a very coarse level. The

Control component, therefore, has to be (re)configured as one unit. This is clearly inefficient as the join and leave procedures have to be reconfigured alongside the maintenance and repair to achieve adaptability. It also creates a burden on developers as large amounts of code have to be repeated in multiple components even when changes only affect very small parts. Further, coarse granularity also creates issues for the decision engine responsible for making component selections. This is because components that possess large amounts of functionality can have elements that are well suited to their environment but also aspects that are not. This greatly complicates the decision process as it now becomes necessary to weigh off the different trade-offs within the components itself. For instance, in Pattern I, a Control component could contain optimal maintenance functionality but ill-suited repair functionality.

The component Pastry implementation, however, achieves the objective effectively. By separating out the maintenance and repair procedures into independent components, the system can now (re)configure itself efficiently without thought to the other aspects, relating to control elements. Further, it is possible to take an even finer grained approach by exploiting independent access to the individual algorithms responsible for each overlay data structure. This allows, for instance, easy adaptation in the routing table whilst not affect the ring topology maintained in the leaf set. As well as this, through the architecture's open event model, it is easy to combine the functionality of multiple components. Therefore multiple Maintenance and Repair components can exist in the architecture, working in cooperation. This allows components implementing new capabilities to augment existing ones without the necessity to repeatedly implement base functionality. Table 5.1 shows the component configurations used to achieve the case study. It is easy to identify obvious configurations; for instance, when operating in the Internet, Pastry uses full leaf set broadcasts to maintain the topology. However, in a campus environment it utilises the lower overhead approach of periodic keep-alive messages as the reliable, low latency nature of the environment makes this sufficient.

Environment	**Configuration**
Internet	*Maintenance:* - Leaf Set Member Broadcast - Lazy Routing Table Failure Discovery *Repair:* - Standard Repair - Local Black-List Repair
Campus	*Maintenance:* - Leaf Set Keep-Alive - Lazy Routing Table Failure Discovery *Repair:* - Standard Repair - Administrator Notification Repair - Centralised Black-List Repair

Table 5.1 Pattern III Case Study Component Configurations

As well as this, more sophisticated configurations can also be utilised. Most notably, it is possible to exploit the combination of multiple components. When operating in the Internet, Pastry utilises two Repair components: Standard Repair and Local Black-List Repair. This latter augments standard functionality by maintaining a black list of malicious and unreliable peers, installing itself above the Standard Repair component

in the event tree. Therefore, on receipt of a routing table failure event, it locates a suitable (non black-listed) replacement before forwarding the event to the Standard Repair component. The Standard Repair component then updates the necessary state entities and notifies the appropriate nodes. This can be contrasted with the campus scenario in which the Standard Repair component is accompanied by the Administrator Notification Repair and the Centralised Black-List Repair components. In this environment, if a routing table failure is detected, the Centralised Black-List Repair component utilises a centralised database to validate the chosen replacement. Similarly, misbehaving peers (e.g. frequent failure and rejoins) are reported through the Administrator Notification Repair component which passively monitors joining, repairing and routing events. This rich variation in functionality is not possible with coarser grained models; this is because it is not possible to 'mix and match' components. Instead, it is necessary to implement a large number of Control components, each containing monolithic variations. This is resource intensive, highly complex and requires intensive coding.

Maintenance Components	Repair Components
Leaf Set Member Keep-Alive	Standard Repair
Leaf Set Member Broadcasts	Administrator Notification Repair
Probabilistic Leaf Set Keep-Alive	Local Black-List Repair
Routing Table Member Keep-Alive	Centralised Black-List Repair
Lazy Routing Table Failure Discovery	Certificate Validation Repair

Table 5.2 Maintenance and Repair Components

The fine-grained nature of Pattern III therefore allows substantial and effective (re)configuration to take place in the overlay. This, when compared to coarser models, can be seen to create strong functional incentives for development in this manner. Therefore, whilst coarser models offer high level adaptive properties and well structured implementations, they cannot support the diversity of environments and requirements that are possible through finer grained models. Table 5.2 shows a number of components that can be utilised with Pastry. These components are capable of supporting a range of constraints and requirements. For instance, low overhead mechanisms can be employed such as lazy routing table maintenance, keep alive leaf set maintenance and the local black listing of peers. However, these can also easily be replaced to provide more reliable support e.g. routing table keep-alive maintenance and administrator notification. As well as this, variations in application level requirements can be easily implemented. For instance, secure and closed networks can utilise certificate validation in the join and repair procedures to only allow validated members. Vitally, such configurations are performed in conjunction with conventional existing, non-modified, components. An alternative evaluation that focuses on (re)configuration can also be found in our existing work [27].

6 Evaluation of Performance and Engineering

Section 5 has provided an evaluation of the potential of functional (re)configuration. We evaluate the approach's non-functional properties based on the following four criteria:

i) *Resource Overhead:* Is the overhead incurred (in terms of performance throughput and memory costs) by fine-grained architectures acceptable?
ii) *Ease of Development:* How easy is it for a developer to create, configure, and extend an overlay?
iii) *Reusability:* To what extent can components developed for a particular overlay implementation be reused?
iv) *Functional Evolution:* To what extent can the overlay evolve to include new functionality?

6.1. Resource Overheads

This section examines the performance overheads associated with implementing an overlay network using components. All tests were performed on a 1.7GHz Intel Pentium M processor; 512 Mb RAM; Sun JVM 1.6.0.1; the components were developed using the OpenCOM v1.4 framework [7]

6.1.1. Throughput Overhead

This section demonstrates the operation call throughput overhead of using components compared to traditional object orientated approaches. This highlights the overhead associated with implementing overlays in a component based fashion. The first experiment is to invoke a null operation (no parameters, and no operational logic to measure maximum overhead impact) 100,000 times on a Java object implementation; this experiment was repeated 5 times and the median value taken. The same procedure was repeated for invoking operations on an equivalent OpenCOM component through a receptacle call. The results of these experiments are illustrated in table 6.1. It can be seen that receptacle calls have a 57% decrease in throughput and are therefore more expensive than object based native method calls. Receptacles, however, reduce coupling in the system and provide support for dynamic evolution and reconfiguration therefore creating a trade-off in performance.

Type	Throughput(Invocations/Second)
Java Method Call	208.768267 x 10^6 (208 million)
OpenCOM Receptacle Call	91.785222 x 10^6 (91 million)

Table 6.1. Invocation Throughput

In finer-grained component architectures where there are a large number of components, there will be an increasingly large number of component interactions required for functions to be performed. Therefore, the effects of component

throughput will be directly based on how many components there are in the system. This, however, is not an issue that should be of concern unless the overlay is required to utilise the maximum operational throughput (~90 million/sec); this has never occurred in our implementations. Further to this, its distributed nature renders the decreased operational throughput as negligible. For instance, when performing a Pastry join over a small network the join time is 10.8 seconds. This will, at most, require 15 component interactions through the event passing framework. This shows that the overhead of component interactions constitute under 0.001% of the overall overhead. Therefore, in a distributed environment, the overhead of using component interactions is insignificant. Further, the ability to streamline and optimise implementations through configuration means that the overall system overhead (e.g. bandwidth utilisation) is decreased.

In the Pastry implementation, control is passed between components using either receptacles or event passing. All state and forwarding interactions were performed using receptacles. Alternatively, the control elements performed all interactions using event passing (although these events are similarly passed through receptacles). Table 6.2 outlines the number of components traversed during negotiations.

	Number of Components Traversed			
Process	**Node 1**	**Node 2**	**Node 3**	**Node 4**
Create New Network	3			
Join Node 2	15	12		
Join Node 3	15	5	12	
Join Node 4	12	5	15	15
Fail Node 1		11	7	12

Table 6.2. Component Event Traversals

To initiate a network (i.e. starting up a new individual node) 3 component interactions are required which can be compared to 0 interactions required by a Pattern I control entity. When another node is then subsequently joined to the network a further 15 component interactions are required by node 1 to deal with the request. This process therefore requires an extra 0.163 microseconds for component interactions, creating 227% extra overhead compared to performing the same operations using native Java interfaces. There is therefore a noticeable overhead involved with increasing the granularity of the component pattern used. However, as the advantages of decoupling these functional aspects are significant, they therefore, if exploited, warrant the increased level in overhead. Further, the distributed nature of interactions means that the decreased operational throughput does not adversely affect the overall system performance.

6.1.2 Memory Overhead

This experiment investigates the static memory footprint of implementing overlay functionality in components when compared to conventional Java objects. For the experiment, six modules have been implemented as both OpenCOM components and

Java objects. These types consisted of modules with increasingly larger numbers of interfaces and receptacles. An interface represents the services that a component can provide whilst a receptacle represents the services that a component requires. The memory footprints of the types were then measured, shown in Table 6.3.

Module	Component (bytes)	Java Class (bytes)	Overhead (bytes)
One (1 intf, 0 recps)	990	623	367
Two (2 intf, 0 recps)	1703	1307	396
Three (3 Intf, recp)	2123	1703	420
Four (1 Intf, 1 Recp)	2999	2051	941
Five (1 Intf, 2 recp)	3299	2051	1248
Six (1 intf, 3 recp)	3555	2051	1504

Table 6.3.Memory Overhead of using Components

Developing a component with no receptacles adds approximately 370 bytes of overhead compared to a conventional Java object, with another 20 bytes for each additional interface. This can be compared to approximately 300 bytes extra for each receptacle. This means that, component based overlay implementations will have a marginally larger memory size compared to monolithic or object oriented developments. This overhead, however, is limited to only a small increment compared to alternative approaches. Further, the ability to construct systems from the minimum number of required components means that the overall memory footprint can be reduced by only distributing and loading the necessary components.

6.2 Ease of Development

One interesting area of investigation is how the use of components affects the development process. This section will look at the amount of coding required and the pros and cons related to component management.

6.2.1 Code Complexity

This section investigates the ease of implementing overlays in a fine grained component architecture when compared with more traditional approaches. This is done because fine grained components involve additional code complexity in the form of dealing with event passing and controlling interactions between components. Another major issue is the occasional requirement for components to repeat functionality to ensure the independence of components. This problem can be rectified through the use of even finer grained architectures that place these shared elements into independent components although this might result in greater complexity.

To evaluate the impact that the use of components has on the system, the Join component has been looked at to test the overhead related to coding the OpenCOM and event based elements of the system. The Join component has 194 lines of overlay

related code in it, including 9 methods responsible for the various aspects of the join operation. This component then has the addition of 3 new component references (receptacles) to enable it to interact with the transport, state and forwarding aspects of the system. Further to this an extra 129 lines of code were then attributed to the OpenCOM related aspects of the class leading to a total of 325 lines of code, creating an increase of 39.69% in code overhead and 4,532 bytes of extra static memory.

Measurement	Chord	Pastry (excl events)	Pastry (inc events)
Classes	5	35	53
Packages	4	10	13
Components	3	16	16

Table 6.4. Component Code Complexity

This clearly shows that providing objects with the added elements required to form event passing components creates a noticeable coding overhead. However it should be noted that this overhead comes in the form of template-like coding consisting mostly of event registration and other such operations.

To better gain an understanding of the overhead involved in development, the fine-grained Pastry implementation is compared to a coarse grained Chord implementation (shown in Table 6.4). It can be seen that a much larger number of components, involving a similarly larger number of classes, are used in the fine grained implementation. This is partially attributable to Chord's relative simplicity when compared to Pastry but can also be attributed to the need to support far more components along with the necessity to repeat certain elements of functionality. It can therefore be derived that the use of the finer grained model introduces a noticeable amount of extra classes and components; however this is obviously traded off against the benefits documented in this paper.

6.2.2 Management and Dependency Complexity

The next type of complexity comes in the form of the overhead of managing a large number of components in the system. The majority of benefits that are gained in the system are achieved through the use of fine grained components rather than coarser entities, however, as the number of components increase, as does the complexity involved in managing and instantiating them. Figure 6.1 shows the primary components (Pattern II) and their inter-dependencies

As shown in Table 6.3 the number of component in the Pastry implementation increases from 3 to 16. These components form a well structured hierarchy installed by a user written script or decision engine, and managed by an event based framework. These components embody small sets of well defined functionality and therefore create an improved management approach for the developer, as processes such as dividing work loads between multiple engineers are made much easier. However, when separating functionality into the intermediate level the chances of requiring the repetition of functionality in multiple components increase.

As well as development complexity there is also runtime complexity as the system will now be required to deal with a larger number of components and any subsequent events that they generate. In a coarse, three component model there will only be three generic event capabilities registered, however by decomposing these into finer grained components, another 15 event capabilities need to be registered and handled. This will create little complexity in terms of runtime overhead but will be more evident in areas such as event management when attempting to ensure that all the required events can be serviced correctly (i.e. dependency management).

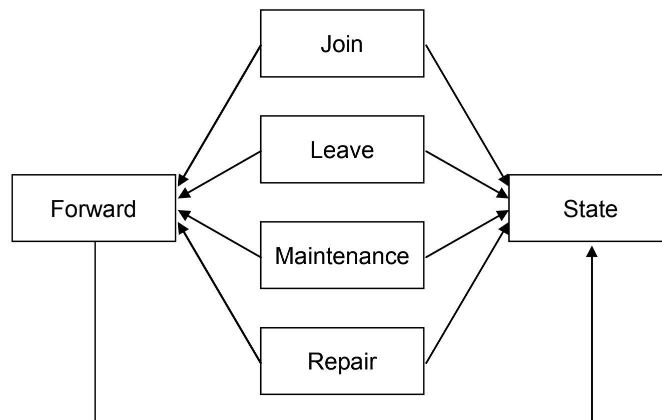

Figure 6.1. Dependencies between Pattern II Components

Further, component compatibility must also be dealt with, as the more complex an architecture is, the harder it is to ensure that all the components are compatible. For instance, Figure 6.1. shows the inter-dependencies between a Pattern II Pastry implementation. Each dependency must be resolved with a compatible components. For Pattern I, the number of dependencies is only 3 when compared with 9 generated by the Pattern II implementation. It is hard for a system to automatically ascertain whether a component is compatible with the system as the concept of black box development makes it hard to analyse how a component works. This therefore means that developers will need to make a concerted effort to ensure that a re-configuration maintains compatibility.

6.3 Reusability

A major benefit of embodying code in components is the possibility to reuse that code without consideration to how it works. There are two case studies that can be looked at in terms of reusability; the first is reusability with overlays of the same type whilst the second is reusability with overlays of different types (portability).

The non-component Pastry implementation struggles to achieve either type of reusability as it has been developed without thought to being used for different systems. Even with the use of effective coding practises it becomes a burden for later

developers to reuse the code as it is necessary to take an introspective view at the source code to derive potentially reusable aspects.

Chord makes substantial improvements in term of conventional reusability as the Forward and State components can easily be reused in other Chord implementations that simply wish to focus on the control aspects. This is because these components play passive roles in the overlay rather than generating dependencies themselves. Reuse of the Forward and State components is also the most likely scenario as there is less utility in modifying these two components. Further to this, the process is simple as there are only two components to deal with. Chord, however, cannot offer any real level of portability as the components are too large to remove small aspects from. Similarly reuse of a monolithic Control component becomes unbeneficial as it is unlikely that developers would not wish to modify the control behaviour in their new overlay. Reuse can therefore only be performed with the same type of overlay.

The component based Pastry is by far the most reusable, offering high levels of reusability alongside a limited degree of portability. The utility of reuse in Pattern III becomes much greater as it possible to reuse much more specific aspects. This allows developers to focus their work on much smaller areas whilst addressing all other issues with reused components. Further, by manipulating events and utilising interceptors between interfaces it becomes possible to reuse and specialise components by simply augmenting existing functionality. Pattern III therefore dramatically improves portability, as things such as the individual maintenance and repair algorithms now become more generic. For example, the maintenance and repair of a Pastry leaf set is comparable to the maintenance and repair of a Chord successor table. Similarly, the state components that store this information can be ported. This is clearly possible due to the topological similarities between Chord and Pastry whereas porting between more diverse overlays such as Pastry and NICE [1] become much more difficult. To achieve this it would therefore be necessary to further increase the granularity of components to encompass and separate individual algorithms.

6.4 Functional Evolution

Another powerful concept is the ability to dynamically extend system functionality; this is done through either adding or replacing components in the architecture. This allows a node to evolve in its environment by obtaining extra components [28].

A case study that offers substantial benefits for an overlay developer is the ability to dynamically extend a node's join operation to be locally aware [5]. This allows a peer to distinguish between nearby and distant nodes. This is highly advantageous especially when dealing with such things as large scale content distribution. This could be achieved by either replacing the join component in the system or alternatively by adding extra components that deal with the locality issues for the other components.

The non-component Pastry implementation cannot deal with these issues effectively. It is possible for a developer to modify the join code but there is no elegant or automated method of deployment. Further to this, the extension of this system would require intricate knowledge of the implementation.

Chord similarly cannot deal with these issues without re-developing the entire Control component. However, the deployment of such an update becomes easier through the use of components as it is now possible to dynamically deploy components between overlay nodes or through automated updates.

The component based Pastry, however, gains substantial functional evolution through its use of Pattern III. By deconstructing the control aspects, developers can gain direct access to the join elements. This allows an independent Join component to be developed and deployed without dealing with other areas. Further to this, even more efficient extensions are possible by simply adding components that deal with the specific aspects of the extension.

The Join component receives events from other components informing it to perform specific operations. This allows new components to be added that deal with a subset of these events. The Join component deals with both, initiating the leaf set which is not locality aware and initiating the routing table which is locality aware. To achieve the case study in Pattern III, the original Join component is left to deal with the leaf set whilst a new component is added to intercept events for the routing table. This becomes possible through component decoupling and the use of shared state components. This is because in Pattern III it is possible for any component to update state information. This allows effective extensions to take place using fine grained state components as a bridge between incompatibilities.

7 Conclusion and Future Work

This paper has investigated the issues surrounding the design, implementation and deployment of peer-to-peer overlays in a fine grained component based fashion. Using the Gridkit Overlay Framework [14] as a nucleus, a component architecture has been developed mandating that overlays are constructed using a particular set of components, to implement the various aspects of functionality resident in the overlay. These components can then be added or removed dynamically in the confines of a framework, to create a flexible, (re)configurable, reusable and extensible overlay.

Four different approaches have been considered ranging from monolithic designs to fine grained component architectures. A number of pros and cons have been identified with each approach; coarser grained components provide a well structured simple approach to overlay design but lack flexibility in terms of reusability, (re)configurability etc. This can be contrasted with fine grained components which offer superior flexibility but with greater overhead.

The investigation has shown that any type of component architecture offers a number of tangible benefits to overlay developers leaving non-component based designs superior only in very simplistic overlays. Coarse grained architectures are simple and provide an adequate model for developing relatively simple overlays with limited levels of re-configurability, reusability etc. It has also been shown that substantial benefits can be gained by utilising finer grained component approaches as proposed in Pattern III. Such an architecture offers a powerful mechanism for the (re)configuration and functional evolution of a system far beyond what is achievable in non-component designs. Further, the advantages gained in reusability and

structured design creates implicit software engineering benefits allowing overlays to be developed in a faster more elegant fashion. An evaluative summary is provided in Table 7.1 with three stars constituting the best score.

*Best Score = * * **	**Monolithic**	**Pattern 1**	**Pattern 3**
Resource Overhead	* * *	* *	* *
(Re)Configurability	*	* *	* * *
Development Complexity	* * *	* * *	*
Reusability	*	* *	* * *
Functional Scalability	*	* *	* * *

Table 7.1. Summary of Evaluative Criteria

There is a variety of future work that can be carried out in this field. The use of component based overlay design has largely been unexplored leaving a number of potential areas of work. So far, the use of DHT systems has been used to investigate the architecture outlined in this paper. It is therefore necessary to expand this work into other areas, such as application level multicast [1], to look at how alternate overlays perform. Further to this, alternative component architectures and even finer grained models should be investigated. Work has already been carried out into identifying fourth tier components including identifier generation, state collection and data dissemination. This work will be continued to look into managing and reusing such fine grained functionality alongside investigating more sophisticated interaction and functional extension techniques. It is hoped that this will lead to the creation of more sophisticated, holistic architectures.

References

1. Banerjee, S., Bhattacharjee, B., and Kommareddy, C. "Scalable application layer multicast". In Proc. ACM SIGCOMM Pittsburgh, Pennsylvania, USA, (2002).
2. Behnel, S., Buchmann, "A. Models and Languages for Overlay Networks". In Proc. 3rd Intl. VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing, Olso, Norway (2005).
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., Saikoski, K., "The Design and Implementation of Open ORB V2". In IEEE Distributed Systems Online (2001).
4. Bruneton, E., Coupaye, T., Leclerc, M., Quema, V., Stefani, J-B, "An Open Component Model and its Support in Java", In Proc. 7th Intl. Symposium on Component-Based Software Engineering Edinburgh, Scotland (2004).
5. Castro, M., Druschel, P., Hu, Y., and Rowstron, A. "Exploiting Network Proximity in Peer-to-Peer Overlay Networks". In Technical Report MSR-TR-2003-82, Microsoft Research (2002).
6. Castro, M., Druschel, P., Kermarrec, A., and Rowstron, A. "SCRIBE: A Large-scale and Decentralized Application-level Multicast Infrastructure." IEEE Journal on Selected Areas in Communications, 20(8):1489–1499, October 2002. Communication, London, UK, November (2001).

7. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, Jo, Sivaharan, T., "A Generic Component Model for Building Systems Software". In ACM Transactions on Computer Systems, 27(1):1-42, February (2008).
8. Dabek, F., Zhao, B., Druschel, P., Stoica, I. "Towards a common API for structured peer-to-peer overlays". In Proc. 2nd Intl. Workshop on Peer-to-Peer Systems, Berkeley, CA, USA (2003)
9. Dowling, J., Cahill, V., "The K-Component Architecture Meta-Model for Self-Adaptive Software", In Proc. 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan (2001).
10. Druschel, P. and Rowstron, A. "PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility". In Proc. 8th Workshop on Hot Topics in Operating Systems Oberbayerrn, Germany. (2001).
11. Fassino, J.-P., Stefani, J.-B., Lawall, J., Muller, G., "THINK: A Software Framework for Component-based Operating System Kernels". In Usenix Annual Technical Conference, Monterey, CA (2002).
12. Foster, I. and Iamnitchi, A. "On death, taxes, and the convergence of peer-to-peer and grid computing". In Proc. 2nd Intl. Workshop on Peer-to-Peer Systems, Berkley, CA. (2003).
13. 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).
14. 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, Cyprus, October (2004).
15. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., Campbell, R.H., "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB". In Proc. IFIP Intl. Middleware Conference, New York, NY (2000).
16. Loo, B.T., Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, T., Stoica, I. "Implementing Declarative Overlays". In SIGOPS Operating Systems Review 75-90 Oct. (2005).
17. Magoutis, K., Brustoloni, J.C., Gabber, E., Ng, W.T., Silberschatz, A., "Building Appliances out of Reusable Components using Pebble". In Proc. SIGOPS European Workshop 2000, Kolding, Denmark (2000).
18. Mathy, L., Canonico, R. and Hutchinson, D. "An Overlay Tree Building Control Protocol". In Proc. of the 3rd International COST264 Workshop on Networked Group Communications, London, UK (2001).
19. Microsoft, .Net Home Page, http://www.microsoft.com/net
20. Plaxton, C. G., Rajaraman, R., and Richa, A. W. "Accessing nearby copies of replicated objects in a distributed environment". In Proc. 9th Annual ACM Symposium on Parallel Algorithms and Architectures, Newport, Rhode Island (1997).
21. RAPIDware Project. Michigan State University, Department of Computer Science and Engineering, http://www.cse.msu.edu/rapidware.
22. Rodriguez, A., Killian, C., Bhat, S., Kosti´c, D., Vahdat, A. "MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks". In Proc. USENIX/ACM Symposium on Networked Systems Design and Implementation, San Francisco, CA (2004).
23. Rowstron, A., Druschel, P., "Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems". In Proc. IFIP Intl Middleware Conference, Heidelberg, Germany (2001).
24. Stoica, I., Morris, R., Karger, R.D., Kaashoek, M., Balakarishnan, H. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications". In Proc. ACM SIGCOMM, San Diego, CA (2001).
25. Sun Microsystems, Enterprise JavaBeans, http://java.sun.com/products/ejb/index.html

26. Szyperski, C. "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, (1998).
27. Tyson, G., Grace, P., Mauthe, A., Blair G, Kaune S. "A Reflective Middleware to Support Peer-to-Peer Overlay Adaptation. In Proc. 9th Intl. IFIP Conference of Distributed Applications and Interoperable Systems, Lisbon, Portugal (2009).
28. Tyson, G., Grace, P., Mauthe, A., Kaune S. "The Survival of the Fittest: An Evolutionary Approach to Deploying Adaptive Functionality in Peer-to-Peer Systems". In Proc. Workshop on Adaptive and Reflective Middleware, Leuven Belgium (2008).
29. Van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. "The Koala Component Model for Consumer Electronics Software". In Computer 33, 3, March, (2000).