The Survival of the Fittest: An Evolutionary Approach to Deploying Adaptive Functionality in Peer-to-Peer Systems

Gareth Tyson[a], Paul Grace[a], Andreas Mauthe[a], Sebastian Kaune[b]

[a]InfoLab21, Lancaster University, Lancaster, UK.

[b]Technishe Universitat Darmstadt, Darmstadt, Germany.

[a]{g.tyson, p.grace, andreas}@comp.lancs.ac.uk, [b]kaune@kom.tu-darmstadt.de

ABSTRACT

The heterogeneous, large-scale and decentralised nature of peer-to-peer systems creates significant issues when deploying new functionality and adapting peer behaviour. The ability to autonomously deploy new adaptive functionality is therefore highly beneficial. This paper investigates middleware support for evolving and adapting peers in divergent systems through reflective component based design. This approach allows self-contained functionality to exist in the network as a primary entity. This functionality is autonomously propagated to suitable peers, allowing nodes to be evolved and adapted to their individual constraints and the specific requirements of their environment. This results in effective functionality flourishing whilst sub-optimal functionality dies out. By this, a self-managed infrastructure is created that supports the deployment of functionality following the evolutionary theory of *natural selection*. This approach is evaluated through simulations to highlight the potential of using natural selection for the deployment and management of software evolution.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications; D.2.11 [**Software Architectures**]: Patterns (Reflection)

General Terms

Design, Management

Keywords

Software evolution, natural selection, peer-to-peer, functional scalability, self-optimisation, reflective middleware

1. INTRODUCTION

Recent years have seen a proliferation in the number of widely deployed distributed systems with a particular focus on peer-to-peer applications. Such systems offer a number of benefits derived from their ability to self-organise and pool resources. Their decentralised nature, however, creates significant issues when managing, deploying and optimising new system functionality.

In a traditional client-server model, introducing new functionality is not a significant issue as administrators can easily update server software and insist that clients do so to gain compatibility. Such an approach, however, is not feasible in a peer-to-peer environment. This is because the lack of centralised management means that functionality can be introduced through individual peers at any time in an uncontrolled way. Further, due to the nature of peer-to-peer networks, specific functionality is not necessarily appropriate for different peers. This means that nodes must be adapted in a very fine grained manner. However, to achieve this it is necessary for peers to be able to evolve their capabilities to address new constraints and requirements. Software *evolution* is the process by which applications can be maintained and extended to incorporate new functionality [11]. Research has largely indicated that the use of well-defined software architectures allows systems to effectively *scale* their functionality through the manipulation of software building blocks.

This form of evolution, however, is limited in scope and does not port well to the concept of fully decentralised systems. This is because it is only concerned with the practical issues of augmenting existing software. This does not take into account such things as the deployment, location or selection of new functionality. In contrast to this, a number of interesting correlations can be drawn between the evolutionary requirements of peer-to-peer systems and those of ecosystems [10]. In natural evolution, organisms that are well suited to their environment flourish and propagate whilst ill-suited organisms die out. This is termed *the survival of the fittest* and has been identified in the software market [7]. This paradigm can offer significant benefits if ported to peer-to-peer systems. Such an approach would allow functionality to autonomously exist in a network in a similar way to life forms in an ecosystem. Through the concept of *natural selection* [10], effective functionality would then survive and propagate whilst poor functionality dies out.

This paper investigates the potential of distributed functional adaptation and evolution in peer-to-peer systems. To this end, an approach is outlined using the Juno middleware [15] as a platform. Through self contained reflective components, users can develop and deploy functionality in a self-optimising and scalable manner. This functionality is then autonomously distributed in the network through the automated evaluation of its performance. Once deployed, functionality can either flourish or die. This results in functionality in suitable environments virally propagating whilst ill-suiting functionality becomes extinct. This approach provides three attractive properties:

    i)   *Autonomous Management* – Peers can inject functionality into the network. If the functionality is effective it will be autonomously distributed.

ii) *Self-Optimisation* – Functionality will naturally be disseminated towards its optimal environments. Similarly, inefficient functionality will die out. This allows peers to be autonomously configured.

iii) *Functional Scalability* – The ability to dynamically exchange functionality ensures that peers supporting the evolutionary platform will always be able to interact by scaling their capabilities through functional exchange.

The paper is structured as follows: Section 2 outlines Juno, the middleware platform used for the system. Section 3 outlines the details of the evolutionary process. Subsequently, Section 4 evaluates this mechanism. Section 5 then provides a brief background to the area. Finally, Section 6 concludes the work, identifying a number of areas of future work.

2. JUNO MIDDLEWARE

To support distributed evolution it is necessary for middleware support to be provided. The evolutionary process described in this paper has been designed to operate with the Juno middleware [15]. This section outlines Juno's relevant operation.

2.1 Overview of Juno

Juno is a (re)configurable peer-to-peer middleware designed to address the heterogeneity of modern content networks [12]. Content networking refers to the progression of traditional content distribution technologies to more integrated, holistic content environments. Unlike traditional content distribution, content networks view the content itself as the focal point of the system. This can be compared to systems such as BitTorrent [4] that view content as just a set of bytes. To this end, content networks utilise information to intelligently distribute content to end users, taking into account such things as user preferences and Quality of Service (QoS). Content networks are therefore often defined by a diverse range of delivery mechanisms and multimedia services, creating significant complexities when evolving applications.
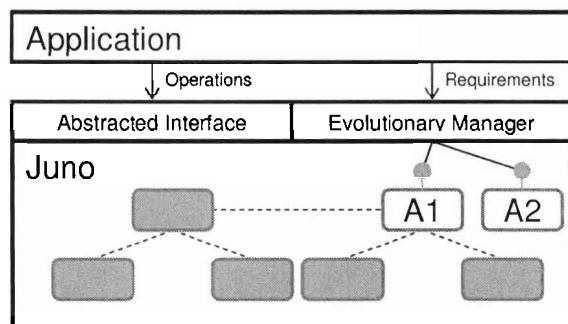

Figure 1 Overview of Juno Middleware (per Host)

Juno's approach to addressing these complexities is to encapsulate functionality in fine grain software components, shown in Fig 1. These are independent software entities that offer abstracted services (interfaces) alongside well defined requirements (receptacles). In Juno, multiple components are interconnected to build multimedia delivery mechanisms and services (e.g. video streaming, transcoding etc). For example, a distributed object location overlay such as Pastry [14] has a number of identifiable functional aspects e.g. joining, maintenance, routing etc. Juno therefore separates these aspects through software modularisation and dynamically interconnects them at runtime to construct a fully operational peer. This allows adaptable systems to be constructed by selectively connecting the optimal components for the particular constraints and requirements the node is operating in.

2.2 Reconfiguration in Juno

The ability to dynamically reconfigure functionality creates a natural platform for adaptation and evolution. Therefore, on receipt of a superior component, Juno dynamically reconfigures its internal architecture to replace the existing component with the new one. For example, a node would evolve its maintenance algorithms by obtaining a new maintenance component that offers the correct interface. The old component would then be removed from the software architecture and replaced by the new one. All subsequent maintenance functionality would then be performed by the new component. To support this, interchangeable components must offer identical interfaces.

2.3 Reflection in Juno

One of the primary functions of Juno is to create a bespoke high level platform for content network applications to operate over. It does this by dynamically constructing itself from optimal components based on environmental factors and application level requirements. To achieve this, however, it is necessary for Juno to ascertain the quality and behaviour of individual components.

To facilitate this decision process, Juno utilises *reflection*. Through the OpenCOM [6] component model, Juno can inspect the operational performance of each available component using quantitative meta-data. Each component implements OpenCOM's IMetaInterface which allows tag based meta-data to be associated with each of the component's interfaces. This is shown in Figure 1 with components A1 and A2 offering the IMetaInterface to the Evolutionary Manager. Each tag represents a particular evaluative metric of the component's performance (e.g. for a caching component, average_hit_rate, storage_overhead etc). To allow comparisons, all components offering a particular interface utilise identical meta-tags. To support this, an application defines standard meta-tags for its *default components*. All future *evolutionary components* must then be described using identical meta-tags to those defined by the default component they replace. This reduces flexibility but is necessary for comparability.

To inspect a component's performance, an application can call the getAttributeValue(String tag) operation on any component to gain a quantitative assessment of a particular facet e.g. the bandwidth_overhead of an overlay maintenance component. This operation returns a quantitative value for the requested meta-tag; the representation and assignment of these values will be looked at in Section 3.

3. EVOLUTIONARY PROCESS

For evolution to take place it is necessary to describe how functionality is propagated in the network. The approach taken follows the theory of the *survival of the fittest*. This section describes how this concept is ported to peer-to-peer environments.

3.1 Evolutionary Dissemination

To allow a fully distributed application to evolve, it is necessary to allow functionality to be disseminated to appropriate hosts. In a peer-to-peer environment frequent interactions occur between nodes in the system. These interactions consist of service requests and provisions. For example, in a peer-to-peer content delivery system, nodes will issue requests to each other for data. These interactions are exploited to exchange reflective meta-data between peers. This is because interacting nodes will often share

similar application level requirements and constraints. Juno therefore monitors the application's interaction with other nodes and subsequently contacts them to offer them new functionality. This approach therefore does not involve any additional overhead for node location or topology maintenance. When two peers interact they concurrently exchange reflective meta-data about any extension components considered to be of interest. Each node then analyses this data in order to select any functionality of interest with evolutionary potential. This therefore allows nodes to flexibly inspect individual attributes considered important for a particular set of individual requirements. This provides support for extremely fine grained evolutionary decisions.

If two interacting peers offer each other similar component functionality and meta-data, these two nodes create a link. These links create clusters, of a limited size, containing peers that have similar requirements and environmental constraints. These are termed *environmental clusters*. If a node locates a piece of effective functionality, it shares it with its cluster. This allows functionality to be quickly disseminated in suitable environments without the overhead of actively locating suitable peers.

3.2 Evolutionary Adaptation

When a node receives reflective information about a new component it is necessary to compare it with the equivalent component it is currently using. To allow this, all meta-values are defined relative to the *default* component. The default component therefore sets a base-line that all evolutionary components are compared against. Therefore if a default overlay maintenance component, on average, generates 10KB overhead per minute and a new overlay component only generates 8KB then its assigned bandwidth_overhead meta-value will be 20%. This is because it improves the overhead by 20%. If, alternatively, a new component generates 12KB then the value will be -20%. This is because it creates 20% more overhead. This approach removes the necessity for other components and applications to possess semantic knowledge of quantitative values. Instead, it is possible to simply consider their capabilities as relative to each other. The assignment of these values will be described in Section 3.3.

Using these meta-tag values, Juno can easily compare multiple components to ascertain the superior choice. To assist in this, an application built over Juno must weight the importance of each meta-tag associated with its constituent components. Both Juno and the application then dynamically modify these values to reflect changes in requirements and constraints. For example, a caching service will place considerable weight on the average_hit_rate meta-tag associated with its replacement algorithm component. However, if the host becomes overloaded it will lower this in favour of decreasing resource consumption.

When an evolutionary component is offered, both the new and old components calculate their *scores* based on the current weightings. This is done by multiplying each meta-tag value by its designated weighting. If a new component achieves a higher score than the existing components then Juno will evolve to incorporate the new functionality. Importantly, a peer can reconfigure itself at any time to utilise any component it possesses. Therefore an old component can be utilised again if it is considered optimal.

3.3 Reflective Meta-Data Assignment

An important aspect of the system is the assignment of values to each meta-tag. It is unwise to allow developers to assign relative values themselves as this is easily open to abuse. Further, this non-adaptive approach will limit the accuracy of values in divergent environments. Instead, as each node operates the component it actively manipulates the relative values to reflect the current experience. It then uses this information when later advertising components to other nodes. Therefore, when a peer receives a new component it takes the existing meta-tag values and incrementally changes them to reflect its experience. The resulting values therefore reflect the aggregated experience of all peers that the particular instance of the component has passed through. As a component penetrates a specific environment these values then become more specialised for that particular environment. If a component does not offer the performance that its meta-data stated then this process automatically rectifies this. Through the adaptive process outlined earlier, the new values assigned to this component will result in it being automatically removed from operation in favour of a superior alternative.

To achieve this measurement process it is necessary for the default components to be bundled with the necessary functionality to measure and allocate meta-values to new components. The process is supported through Juno's open architecture. This allows components to easily monitor each other. The measurement functionality associated with the default component therefore passively monitors all components involved in the application. This is done through open state monitoring (ability to inspect component state) and open event/interface monitoring (ability to inspect component interactions). For example, a default component can measure the latency between nodes by listening to the interactions of the networking components. This, therefore, does not require evaluative information to be provided by the components that are under inspection, mitigating the potential for biased decisions or malicious interference. However, further investigation of this is an important area of future work.

3.4 Survival of the Fittest

Once a peer has identified a new component as a good candidate for evolution it will request it and reconfigure itself. However, it is also important that ineffective components are removed from the network. This improves performance and overhead by ensuring poor functionality is not advertised and exchanged in the evolutionary process. To achieve this, each node is restricted to maintaining a limited set of instances for each component type e.g. Pastry maintenance components. Once this set has reached capacity, the worst performing component is removed to make room for the new one. This results in a situation in which components existing in ill-suited environments die whilst components in well-suited environments virally propagate.

4. EVALUATION

To evaluate the system a simulator has been developed. The simulator operates a peer-to-peer video streaming application based on measurements taken from an existing Video on Demand system [16]. Nodes interact with each other based on this application. These interactions are utilised by the evolutionary process to exchange reflective meta-data about new functionality.

Nodes are bootstrapped in one of a number of possible environments. An environment consists of all peers in the system operating with the same type of device and connectivity. Two video streaming system variants are considered: a relatively homogeneous system (5 environments) and a heterogeneous system (15 environments). A *homogenous* system operates over a limited set of network connections and devices (e.g. PCs, laptops, DSL, wireless etc). Alternatively, the *heterogeneous* system has a much greater range of environments (e.g. PCs, laptops over DSL,

T1/T3 etc; PDAs over wireless and Bluetooth; mobile phones over GPRS, UMTS and Bluetooth; TVs over Cable and DSL). Each of these devices and connections has different requirements. For example, a mobile phone will require sources providing low computation decoding; something that will not affect PCs. Due to space constraints details are not provided of individual component specifications or meta-data. The distribution of nodes in these environments is modelled using a Zipf distribution [1] with PCs (over DSL) constituting the greatest number of peers and PDAs (over Bluetooth) constituting the least.

Parameters	Values
Number of Nodes	20,000
Number of Environments	5 / 15
Number of Evolutionary Components	10 / 25 / 50
Node Distribution per Environment	Zipf ($\alpha = 0.5$)
Maximum Number of Stored Components	5 per node
Environmental Cluster Size	16
Number of Injection Points	8 (Random peers)
Request Distribution	Poisson ($\lambda = 0.693$/sec)
Number of Node Interactions per Request	8

Table 1 Default Parameter Configuration

At bootstrap each node possesses the default *source selector*. Subsequently, a number of new source selector components are developed and injected at random points in the network. The source selector component decides which peers should be used to download video data from. A number of variants therefore exist including latency, bandwidth, monetary and encoding preference mechanisms; active and passive probing mechanisms; gossip-based knowledge sharing and social preference mechanisms. Peers therefore try to gain their optimal source selector through Juno's evolutionary mechanism. Before execution, the simulator allocates each node an ordered list of component rankings. These rankings represent the suitability of each component for the environment that the node operates in (position 0 is optimal). Using this, the simulator measures the performance of the mechanism by inspecting the effectiveness of the decisions taken by each node. This section will inspect the results based on the number of optimisations, the distribution of these optimisations and, finally, the extinction of component in the system. The default simulation parameters are in Table 1.

4.1 Optimisation Levels

An important evaluative metric is how many nodes in the network gain their optimal source selector component. Simulations have first been performed with five environments over 48 hours, shown in Figure 2. Deployments of 10 and 50 new components are shown. Optimisation is fast with a small number of components (10), with 90% of nodes self-optimising after 17 hours. However, even with high numbers of components (50), this is only extended by 5 hours. The final 10% of optimisations, however, is significantly greater in both systems; this is termed the *tailing off* period.

The reason for this decrease in gradient is the existence of *fringe* peers that reside in small environments with few communications. Such peers therefore rarely come into contact with similar peers and fail to construct adequate environmental clusters to gain rare (yet optimal) functionality for their individual requirements. For

both 10 and 50 deployed components, the speed of optimisation begins to noticeably slow once the majority (85%) of peers have optimised. The final 15% of peers therefore constitute the fringe.
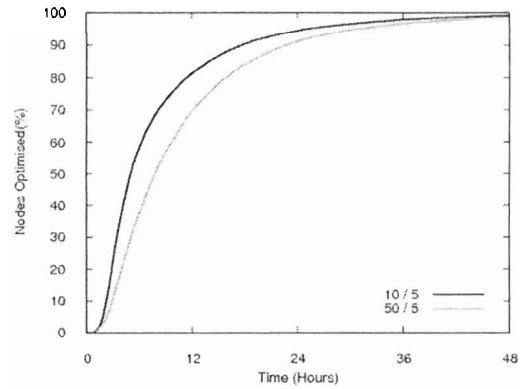

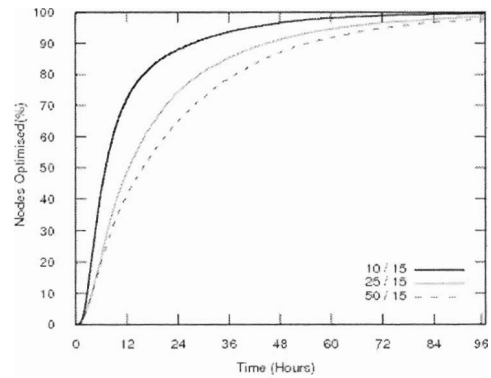Figure 2 Percentage of Optimisations with 5 Environments

Figure 3 Percentage of Optimisations with 15 Environments

Figure 3 shows the percentage of optimisations when operating a heterogeneous system with 15 environments over 96 hours. These experiments highlight the scalability of the approach. When compared to the homogenous system, it can be seen that the speed of optimisation slows down. Further, the tailing off process can also be observed in the same manner as highlighted in the homogenous system. When deploying 10 components, this effect is least noticeable with significant slowing only occurring after ~85% of optimisations. Conversely, when deploying 25 and 50 components, the tailing off procedure occurs in a far smoother manner beginning after ~75%. Importantly, as the number of components grows this tailing off procedure stays fairly constant. Further, the speed of optimisation only marginally decreases. For example, when the number of components increases from 10 to 25, the time taken to reach 90% optimisation increases by 17 hours. However, this can be compared to an increase of only 9 hours when increasing component numbers from 25 to 50.

When comparing the heterogeneous and homogenous systems it can be observed that increasing the number of environments and components only slows the optimisation process; it does not prevent it. For example, after 8 hours, approximately half of all nodes have self-optimised in the homogenous system when deploying 25 components. This can be compared to only 32% in the heterogeneous system. This highlights the complexities encountered when deploying large numbers of components to

many different divergent environments. As both systems enter their tailing off period, however, this difference considerably decreases. After 48 hours, there is only a 7% difference in the level of optimisation between the homogenous and heterogeneous systems (98%, 91%).

This data shows that even when deploying large numbers of components in different types of networks it is possible to effectively evolve functionality in a fully distributed way. The speed of this process is dependent on the number of environments and components. However, the data shows that increasing the number of components does not have a significant impact on the overall optimisation time. Further, even when operating in diverse sets of environments this process can effectively be carried out.

4.2 Environmental Penetration

It has been shown that a significant proportion of the peers are able to self-optimise through Juno's evolutionary process. It is important, however, to investigate the distribution of optimised nodes in environments. Environments with a large number of members (e.g. PCs over DSL) find it easy to gain a high degree of penetration. However, fringe environments with few constituent members (e.g. PDAs over Bluetooth) are less susceptible to fast functional penetration. This is because their limited number of members makes it less likely for a node to interact with another peer possessing the required functionality. To investigate this, simulations are performed to monitor the number of nodes from each environment that optimise. Figure 4 shows each environment's percentage deviation from the overall average percentage of optimisations. These are performed in a heterogeneous system (15 environments) deploying 25 evolutionary components.

It can be seen that early after the components' deployment the deviation between different environments is noticeable. Environment 1 constitutes the largest environment whilst Environment 15 is the smallest. After 48 hours, significant deviations are still identifiable; this is because the larger environments have gained high penetrations whilst the smaller environments have gained lower penetrations. However, after 72 hours these deviations have decreased substantially. For example, between 48 and 72 hours, Environment 7 improves its deviation from -33% to -16%. These experiments corroborate the earlier optimisation experiments, highlighting the difficulty in penetrating small fringe environments. A downward trend can therefore be identified with Environment 1 gaining high levels of penetration whilst Environment 15 gains lower. Interestingly it is also identifiable that some more populous environments (e.g. 7) gain lower penetration than some less populous environments (e.g. 15). This is due to the passive nature of node discovery i.e. evolutionary interactions are based on the higher level application. Therefore, some environments can gain better penetration because their constituent nodes interact more frequently. The environmental penetration is therefore based, more specifically, on the number of interactions; something which is clearly an artefact of the application that is operating. This is an interesting observation that will form a body of future work.

As the time after deployment increases all the environments become closely inline with the average optimisation level. Therefore the deviation of populous environments reduces (e.g. after 96 hours, Environment 1 has a 0% deviation) whilst the less populous environments gain higher levels of penetration (e.g. after 96 hours, Environment 8 only has a -6.5% deviation).
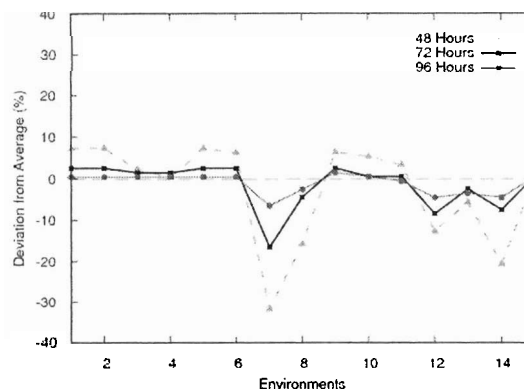

Figure 4 Deviation from Overall Average Percentage of Optimal Nodes for each Environment

These experiments have highlighted the complexities in penetrating small environments. However, the experiments have proved the system to be capable of effectively evolving functionality when operating in a large number of environments.

4.3 Functional Extinction

To accurately reflect the evolutionary process it is necessary for suboptimal functionality to die. However, it is important to ensure that functionality is not removed before being given the opportunity to reach its optimal environments and flourish.
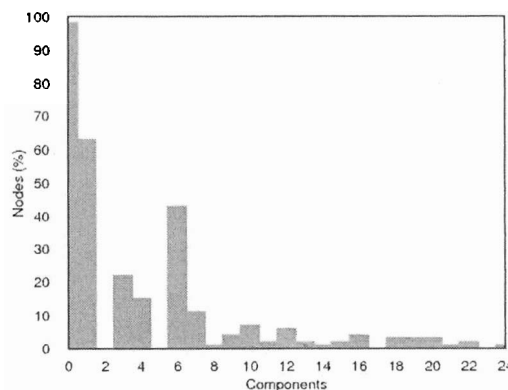

Figure 5 Percentage of each Component in Environment 0

To investigate the death of functionality, the most populous environment is inspected in a heterogeneous system with 25 injected components. Figure 5 shows the percentage of nodes in this environment possessing each of the 25 components after 48 hours. Component 0 represents the optimal source selector for the environment whilst Component 24 represents the least effective. 99% of all nodes in this environment are optimised with Component 0. Further, 63% of the nodes possess the component ranked as the second most optimal. However, very few nodes possess the lower ranked components. For instance, Component 24 is possessed by only 1% of peers. This represents the extinction of functionality in particular environments. It should be recognised, however, that this functionality flourishes in other environments. For instance, there are no instances of Component 17 in Environment 0. However, 99% of PDAs connected through wireless Ethernet possess the component. This shows the system's ability to remove poor functionality from an environment whilst propagating it in other environments.

5. RELATED WORK

A range of work has been carried out into various types of *software evolution*. Early work such as [11] looked at evolving software for maintenance purposes. This promoted the use of architectural patterns and component interceptors to dynamically expand functionality through component models such as OpenCOM [6]. This, however, only deals with internal software concerns rather than deploying and adapting remote nodes. A biological view of software evolution can also be found in [10].

Hales et al [9] utilise an autonomic approach to the evolution and selection of peer-to-peer protocols by inspecting the 'utility' of other peers using alternate protocols. This, however, does not support fine-grained service composition as our approach does. Nor does it allow the exchange of reflective information to permit flexible decisions based on individual node preferences. Instead, only a single universal utility value is exchanged.

To support evolution a number of other systems utilise *reflection* which is the ability for a system to reflect on its own operations and behaviour. This has been exploited in such systems as [13] to allow software to effectively evolve. There are also middleware, such as QuA [8], RAMES [5] and [2] that specifically attempt to address system evolution. These middlewares, however, consider the *local* extension of functionality rather than the distributed aspects investigated in this paper. To address this, mobile agents have been used [3]. However, agents must be constructed on centralised servers and do not support fine grained evolution for peers in different environments. To the best of our knowledge, this work is the first to look at a natural selection approach to disseminating evolutionary functionality in heterogeneous, decentralised environments.

6. CONCLUSION AND FUTURE WORK

This paper has investigated the potential of large scale functional evolution in peer-to-peer systems through the paradigms of natural selection [10] and the survival of the fittest [7]. Based on this, an approach using the Juno middleware [15] has been designed and evaluated. In this approach, evolutionary functionality is encapsulated in reflective components that are exchanged by peers based on their performance and capabilities. This results in effective functionality flourishing in desirable environments whilst poor functionality dies out. Through extensive simulations, the approach was shown to perform well in a VoD scenario, allowing peers to be evolved and adapted in a fine grained manner.

A number of areas of future work can be identified. A major area of further investigation is the system's performance when used with a variety of applications. The evolution of multiple cooperating components is also an important area. The evaluation will be continued to involve these concerns alongside the integration of dynamic environmental changes. Further, the effects of progressive deployments of components will be looked at. Lastly, it is also important to investigate the security of the system by protecting is against the propagation of malicious functionality.

7. ACKNOWLEDGEMENTS

8. REFERENCES

[1] Adamic, L.A. and Huberman, B.A. Zipf's law and the internet. Glottometrics 3 p143-150 (2002).

[2] Arcelli, F. and Raibulet, C. Evolution of an Adaptive Middleware Exploiting Architectural Reflection. In Proc. ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, Nantes, France (2006).

[3] Bettini, L., De Nicola, R., and Loreti, M. Software update via mobile agent based programming. In Proc. of ACM Symposium on Applied Computing, Madrid, Spain (2002).

[4] BitTorrent Specification. http://www.bittorrent.org/beps/bep_0003.html.

[5] Cazzola W., Ghoneim, A., and Saake, G. RAMSES: a Reflective Middleware for Software Evolution. In Proc. of ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, Oslo, Norway (2004).

[6] Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J. and Sivaharan, T. A Generic Component Model for Building Systems Software. In ACM Transactions on Computer Systems, 27(1):1-42, February (2008).

[7] David, J.S., McCarthy, W.E., and Sommer, B.S. Agility: the key to survival of the fittest in the software market. Communications of ACM 46, 5 p65-69 May (2003).

[8] Eliassen, F., Gjørven, E., Eide, V.S., and Michaelsen, J.A. Evolving self-adaptive services using planning-based reflective middleware. In Proc. of Workshop on Adaptive and Reflective Middleware (2006).

[9] Hales, D. and Babaoglu, O. Towards Automatic Social Bootstrapping of Peer-to-Peer Protocols. In ACM SIGOPS Operating Systems Review vol. 40, no. 3, July (2006).

[10] Hutchins, D. A Biologist's View of Software Evolution. In Proc. ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, Glasgow, UK (2005).

[11] Oreizy, P., Medvidovic, N., and Taylor, R.N. Architecture-based runtime software evolution. In Proc. of Intl. Conference on Software Engineering Kyoto, Japan (1998).

[12] Plagemann, T. Goebel, V., Mauthe, A., Mathy, L., Turletti, T. and Urvoy-Keller, G., From Content Distribution to Content Networks – Issues and Challenges. Computer Communications, vol. 29, issue 5, pp. 551-562 (2006).

[13] Rank, S. Architectural reflection for software evolution. In Proc. ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, Glasgow, UK (2005).

[14] Rowstron, A. and Druschel, P. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In Proc. of Middleware, Heidelberg (2001).

[15] Tyson, G., Mauthe, A., Plagemann, T. and El-khatib, Y. Juno: Reconfigurable Middleware for Heterogeneous Content Networking. In Proc. Intl. Workshop on Next Generation Networking Middleware, Samos Islands (2008).

[16] Yu, H., Zheng, D., Zhao, B. Y., and Zheng, W. Understanding user Behavior in Large-Scale Video-on-Demand Systems. In Proc. of ACM Sigops/Eurosys European Conference on Computer Systems (2006).