Self-Organization in Overlay Networks

Sven Apel[1] and Klemens Böhm[2]

[1] Otto-von-Guericke-University, Magdeburg, Germany
`apel@iti.cs.uni-magdeburg.de`
[2] University of Karlsruhe (TH), Germany
`klemens.boehm@ipd.uni-karlsruhe.de`

Abstract. Overlay networks are an important kind of P2P infrastructures. The range of applications and requirements is broad. Consequently, our research objective are overlay networks which organize and adapt themselves at runtime. This article describes the current state of our project and gives an overview of the steps envisioned. We briefly show the necessity for self-organization in overlay networks. Based on our experience, we then provide a list of overlay network system parameters relevant for dynamic adjustment. When designing mechanisms for self-adaptation for overlay networks, we have observed *implementation-level interference* and *semantic-level interference*. To deal with these phenomena, the overlay network architecture envisioned (1) must separate functionality for self-organization and system core functionality and (2) should preserve system integrity. To achieve this, we propose to use self-tuning mechanisms with explicit pre- and postconditions and conflict resolution as well as reflection. We discuss alternative implementation techniques and present one concrete approach based on Aspect-Oriented Programming and Mixin Layers. We conclude with first insights into organic overlay networks and emergent behavior.

1 Introduction

The Peer-to-Peer (P2P) paradigm will gain momentum in the near future, due to scalability, reliability and (economic) efficiency. *Distributed hash tables (DHT)* follow this paradigm: They store and administer huge sets of (key, value)-pairs in a highly distributed manner, without any central coordination. The virtual network structure of DHTs is independent of the underlying physical network. Therefore, they are also called *overlay networks*[3]. Overlay networks have many potential applications in the area of the WWW and Internet computing [10]. Based on our experience [4, 5], this article starts by explaining why overlay networks must be able to adapt themselves at runtime. In short, there is a broad range of application requirements as well as characteristics of the environment that may change over time. This calls for self-adaptation/self-organization. This article describes the current status of our project aiming at self-organizing overlay networks and the steps envisaged.

[3] In the remaining article we use the term 'overlay network' to refer to this class of distributed systems.

In the absence of a coordinator, runtime adaptation is not trivial. Each peer must adapt its behavior individually, but in concert with other peers. When we speak of self-adaptation of an overlay network we mean the adaptation of the participating peers. We refer to the entire process of self-adaptation as *self-organization* or, synonymously, *self-tuning of the P2P system*. Our previous attempts to implement such self-tuning mechanisms have revealed the daunting complexity of this task. The more mechanisms we have integrated, the more they interfere with each other and with the core functionality. Interference may take place at the *implementation level* and at the *semantic level*. These two kinds of interference are different, as we will explain.

This paper discusses the applicability of various software-engineering principles, notably reflection, aspect-orientation, components and meta-data techniques, to implement self-tuning for overlay networks in a modular and efficient manner. We propose a concept for a reflective framework on top of overlay networks. This solution (1) must separate functionality for self-organization and system core functionality and (2) should preserve consistency using conditions. To achieve this, we propose to use self-tuning mechanisms where pre- and post-conditions are made explicit as well as reflection. We sketch three potential ways of implementing self-tuning mechanisms inside the reflective framework and discuss their pros and cons, based on the current state of our project. Furthermore, we present one concrete implementation approach based on AOP and Mixin Layers. Finally, we discuss preliminary results regarding organic overlay networks and emergent behavior.

The remainder of this paper is as follows: Sections 2 introduces overlay networks. Section 3 reviews relevant software-engineering methods. Section 4 introduces self-tuning in overlay networks and describes open problems. Section 5 presents a concept for a reflective framework on top of overlay networks and describes the integration of self-tuning mechanisms. Section 6 discusses open questions regarding the implementation. Section 7 presents a first approach for implementing self-tuning mechanisms. Section 8 outlines future work and Section 9 concludes.

2 Overlay Networks and DHTs

Most of the current overlay networks constitute DHTs. The structure of the data administered by overlay networks is simple, namely (key, value)-pairs. Each peer administers some of these pairs. A hash function, applied to the keys, distributes the data over the peers. Overlay network variants differ in the topology of the key space and in the definition of proximity. *Chord* [26] organizes the key space in circular one-dimensional vector space. *Pastry* [23] uses a Plaxton Mesh. Overlay networks provide an interface identical to the one of an ordinary hash table.

Content-Addressable Networks (CAN) [20] are another important kind of an overlay network, and we have gained experience with overlay networks by implementing a CAN [4]. The key space with CAN is a n-dimensional coordinate space. Each peer is responsible for a zone of the key space. This means that

external characteristics	
network latency	(*nl*)
transfer rate	(*tr*)
network topology	(*nt*)
connectivity	(*co*)

Table 1. External system characteristics.

internal characteristics	
# peers	(*np*)
workload	(*wl*)
data density	(*dd*)
% unreliable peers	(*up*)

Table 2. Internal system characteristics.

the peer stores all (key, value)-pairs whose keys fall into its zone. A query is evaluated by forwarding messages from peer to peer in a greedy fashion, until a peer responsible for the query point is reached [20].

The range of applications for overlay networks is broad, see [10] for a comprehensive list, and our objective is the design and implementation of an overlay network that supports most of these. Overlay network applications have requirements that are quite diverse and change over time. For example a mobile groupware demands for a secure and reliable communication and has to cope with resource-constraints and mobility. Instead, a web annotation service requires a flexible load balancing and a mechanism for a fair distribution of work [4]. In order to fit these requirements and to face environmental changes, e.g., network traffic, workload, connection interrupts, available resources, overlay networks have adapt and organize themselves dynamically at runtime. In fact that means that the individual peers have to alter their behavior.

Our experience with the two mentioned applications, the groupware for mobile ad-hoc networks and the web annotation service, has revealed that the number of *system characteristics* that influence the behavior of overlay networks is high. We distinguish between *external characteristics* and *internal characteristics* (cf. Tables 1 and 2). External characteristics are characteristics of the environment, internal ones are environment-independent. Furthermore, we see several *indicators* which allow a peer to derive the current status with regard to these characteristics. For example, the number of incoming queries per time serves as an indication of the current workload. Table 3 lists some indicators and the corresponding characteristics. To behave in a way that is acceptable for the application, overlay networks have to adapt to changes of the environment. We have identified several tuneable *system parameters*, listed in Table 4, together with the indicators they can react to.

Example. A peer detects an increase of the workload by observing indicator 'rate of incoming queries'. If the rate of incoming queries exceeds a threshold, the peer can try to increase the degree of replication, next to other options. Then the number of peers processing the queries increases. Hence, the rate of queries processed by the 'original' peer decreases. As a side-effect, availability of data increases as well. In contrast, if the rate decreases, the replication degree can be reduced, to save resources and to reduce the overhead of replica maintenance. □

With conventional overlay networks, participation in the work is voluntary. I.e., peers are free to decide how many resources they provide to the overlay

indicators	**characteristics**
⊘ query response time (t_{resp})	*nl*, *tr*, *nt*, *wl*, *co*
# unanswered queries (q_{answ})	*up*, *co*
# forwarded messages (m_{forw})	*np*, *wl*, *dd*
# incoming queries / time (q_{inc})	*nl*, *tr*, *np*, *wl*, *dd*

Table 3. Relationship between indicators and system characteristics.

indicators	**parameters**
q_{answ}, t_{resp}	size of contact cache
t_{resp}, m_{forw}, q_{answ}	replication strategy
q_{inc}, q_{answ}, m_{forw}	replication degree
q_{answ}, t_{resp}	# communication channels
q_{answ}	reliability threshold

Table 4. Relationship between tuneable parameters and indicators.

network. Obviously, this soon leads to *free riding*. Free riders are peers that try to benefit from the system, i.e., query the overlay network, but do not participate in the work, i.e., forward or answer queries. We also refer to free riders as *unreliable peers*. Various approaches against free riding have been proposed, e.g. quorum-based and reputation-based schemes [7]. We for our part have implemented a lightweight reputation mechanisms for CAN [4]. It will serve as an example of some of the issues in the following, so we review it: A peer generates positive feedback on a peer that has performed useful work. A peer may also generate negative feedback in certain situations. Feedback is distributed in a swarm-like manner. It is time stamped, in order to expire after some time. A peer uses the feedback stored in its *reputation repository* to decide if it deems another peer reliable – the number of positive feedback objects must be higher than the *reliability threshold*. A peer ignores feedback attached to an incoming message if it does not deem the sender reliable. – This scheme is a self-tuning mechanism that determines behavior and performance characteristics of the overall overlay network. It is not part of the core functionality – some applications/contexts do not need it, e.g., groupware in local area networks.

3 Software Engineering Methods for Self-Organization

For implementing self-adaptive overlay networks the following paradigms are important:

Reflection. Reflection is one widely accepted method to implement self-introspection and self-adaptation [14]. Reflection was first introduced in the context of programming languages [25]. Reflective programming languages enable the programmer to reason about their language constructs, e.g., classes, methods, etc., and to observe and modify them. Later, this concept has been applied to software systems in general: Reflective software systems make their structural and behavioral properties explicit: They are able to reason about themselves (*introspection*) and to alter their structure and behavior (*adaptation*). To do so, they refine their internal structure and behavior in form of meta-data. The spectrum of possible adaptation mechanisms reaches from simple parameter adjustments, e.g., increasing a value, to complex structural changes, e.g., modifying several methods and classes. Both theoretical [25, 15, 14] and practical work [22, 9] has been done in the area of reflective systems. But to our knowledge, there has not been any research in reflective P2P systems. On the other hand, systems as *BeeHive* [18] and others [19] do focus on dynamic adaptation, but only on the protocol level.

We argue that a P2P system can benefit from reflection by observing its internal state, e.g., the current workload, and by reacting by invoking a load balancing mechanism. Note, the introspection and adaptation is done by the individual peers. There is no centralized view of the overall P2P system. Currently, it is unclear how can a reflective architecture be implemented on top of overlay networks? Which software engineering methods help to implement a reflective P2P architecture?

Separation of Concerns. To implement self-organization, the paradigm *separation of concerns* is essential [8]. The idea is to separate the basic functionality from special-purpose concerns (e.g., synchronization, real-time constraints). [8] argue that this separation makes programs easier to write and to modify. This separation is important for self-organization of overlay networks – our experience shows that the self-organization mechanisms are strongly tangled with the core functionality: In order to implement our light-weight reputation mechanism (see Section 2), we had to modify five classes at 17 code positions and we had to implement five new classes. These numbers tell us that we should not aim for self-organization in our context without proper software-engineering support. Further, the problem becomes more daunting if more self-tuning mechanisms are applied.

Aspect-Oriented Programming. *Aspect-oriented programming (AOP)* [13] is strongly related to reflection and is one approach to support separation of crosscutting concerns. The idea behind AOP is to implement crosscutting concerns as *aspects*. This prevents the known phenomena of code tangling and scattering. The core features are implemented as components, as with common design and implementation methods. Using *pointcuts* and *advices*, an *aspect weaver* brings aspects and components together. Pointcuts specify the join points of aspects and components. Advices specify which code is applied to these points. *AspectJ*[4] is a language extension for compile-time aspect weaving. [17, 27] propose and investigate dynamic weaving mechanisms. Dynamic weaving is more flexible, but consumes more resources.

AOP is a candidate to implement self-organizing overlay networks: AOP enables the programmer to separate the introspection and adaptation code from

[4] http://eclipse.org/aspectj/

the peer core. This prevents the above mentioned code tangling and scattering. Furthermore, AOP allows to flexibly alter self-tuning mechanisms by weaving new aspects at compile time or at runtime.

Component Techniques. Component models like *Mixin Layers* [3] facilitate modularization of crosscutting concerns at class-, object- or method level by composition of components. Mixin Layers are one appropriate technique to implement software systems in a feature-oriented fashion [24, 3]. The basic idea is that features are often implemented by a collaboration of class fragments. A Mixin Layer is a static component encapsulating fragments of several different classes (Mixins) so that all fragments are composed consistently. Mixin Layers are a well-established implementation technique for component-based layered designs. Advantages are a high degree of modularity and easy composition [24]. *AHEAD* generalizes the concept of Mixin Layers to all kinds of software artifacts, e.g., documentation, UML diagrams, and makefiles.

An overlay network product line based on AHEAD could be used to derive tailor-made overlay networks in order to fit application requirements or specific environments. The *AHEAD design rule checks* [3] would allow semantic checking during the composition of overlay networks from different components.

4 Self-Tuning in Overlay Networks

This section describes problems related to self-tuning overlay network. A *self-tuning mechanism* observes system characteristics (see Tab. 1, 2) using indicators (see Tab. 3) and adapts system parameters (see Tab. 4) related to these characteristics. The two self-tuning mechanisms described next serve as a running example throughout this article. They are simple and unexciting, but this should allow for a better focus on issues that are essential.

Number of incoming queries. The number of incoming queries is an indicator. We can infer from it the average computational load and the network traffic inside the overlay network. With some overlay networks, if the number of queries exceeds a threshold, a peer can replicate parts of its data repository to other peers. In other words, the corresponding system parameter in this example is the degree of replication.

Network transfer rate. The network transfer rate is a more technical system characteristic. If it decreases dramatically, or if the connection is interrupted, a peer can use alternative communication protocols or media. The corresponding system parameter in this example is again the replication degree: Increasing it increases availability of the data and allows to cope with changes of the transfer rate.

We have identified self-tuning in overlay networks as a crosscutting concern. It is not trivial to integrate several self-tuning mechanisms, due to their crosscutting nature. Their implementations interfere with each other and with the implementation of the peers core functionality. If we want to apply the example mechanisms to an overlay network, we have to add code for introspection of the system characteristics (transfer rate and workload). This code is tangled with the code for 'core' communication, e.g., listing and accepting connections and receiving messages. Further, several classes and methods have to be implemented to analyze changes of the characteristics. In addition, the replication code itself crosscuts wide parts of the core functionality. The same problems have occurred when integrating our lightweight reputation mechanism (see Section 3). We had to modify numerous implementation units (5 classes, 17 code points, 5 new classes). In the remaining article, we refer to these effects as *implementation level interference*.

Further, the problem of *semantic interference* arises. Semantic interference means that two mechanisms conflict and work in the opposite direction. Think of our examples: When integrating the two self-tuning mechanisms into the peers, two conflicts can occur: (1) The current system configuration does not include replication components. (2) The two mechanisms adjust the same system parameter, the replication degree.[5] One mechanism may increase it, while the other one decreases it. Even if the overall system behavior is meaningful, two concurrent mechanisms waste resources, or the system state becomes instable.

Summing up, the following challenges arise when implementing self-organizing overlay networks: How to implement self-tuning mechanisms in a modular way? Is it possible to apply several mechanisms without crosscutting each other or the system core? How to avoid semantic interference?

5 Reflective Self-Tuning Overlay Networks

This section introduces a reflective architecture for overlay networks, in order to integrate self-tuning mechanisms. We list and discuss alternative methods to add self-tuning mechanisms to the reflective framework. While these methods are standard, our discussion of their applicability for overlay networks is original.

5.1 Reflective Architecture

Figure 1 depicts the overall architecture of the system envisioned, including the system core, the reflective framework and the plugged self-tuning mechanisms.
Any reflective architecture separates the base level from the meta level and reifies meta data. One specialty of the architecture proposed here is the encapsulation of self-tuning mechanisms into pluggable units. The plugged self-tuning mechanisms operate on the reified meta data to alter the peers structure and behavior. We do so to avoid interference (semantic and implementation level). To encapsulate self-tuning, we intend to use components or component-like entities, e.g., aspects. Recall that components are large-scale software building blocks that can encapsulate even complex self-tuning mechanisms. This avoids the interference problem at the implementation level. To avoid interference at

[5] Note that we refer to the plugging of two mechanisms to one peers instance.

the semantic level, we intend to integrate conditions into the self-tuning mechanisms (see Fig. 1). Similar approaches exist in the domain of component techniques, e.g., AHEAD design rule checks. We want to investigate how to reuse these approaches for overlay networks. Similar to AHEAD design rule checks we distinguish between pre- and postconditions. Preconditions state prerequisites to apply the self-tuning mechanism, postconditions state which conditions hold after the self-tuning mechanism has kicked in.

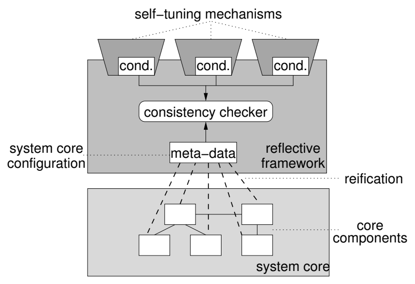

Fig. 1. A reflective overlay network

Example. Think of our self-tuning mechanisms 'number of incoming queries' and 'network transfer rate'. Plugging both may cause conflicts: (1) The core has no replication entity. (2) Both adjust the same parameter, namely the replication degree. Pre- and postconditions avoid these conflicts: (1) Each mechanism defines a precondition to state that it requires a replication entity in the core. (2) The mechanisms provide a postcondition that declares that they adjust the replication degree. The reflective framework can detect these conflicts by processing the conditions. □

The reflective framework manages the self-tuning process. It provides an interface for the application programmer to add, remove or configure self-tuning mechanisms. Moreover, it provides (reifies) information on the structure of the underlying system and of the current system state and behavior. Candidates for reification in our context are the characteristics from Tables 1 and 2, the indicators from Table 3, and the parameters from Table 4.

As a subsequent step, the *consistency checker* compares the pre- and postconditions to the internal representation of the system core. If it detects conflicts of any of the two kinds, it informs the application using a callback mechanism.

6 Implementation Issues

This section discusses alternatives to reflect modifications of reified parameters by the self-tuning mechanisms into the peers, namely *aspect-oriented programming*, *component techniques* and *architectural reflection*. After describing these alternatives briefly, we talk about their strengths and weaknesses.

Aspect-Oriented Programming. One can use AOP to weave dynamically the code of the self-tuning mechanisms to points *inside* the components of the system core. Figure 2 illustrates this. The self-tuning mechanisms provide aspect implementations (dark gray box) written in a specific aspect language. The configuration unit uses an aspect weaver to weave them into the core components of the overlay network (bent arrows). Depending on the plugging method, the weaving takes place at load time or during runtime.

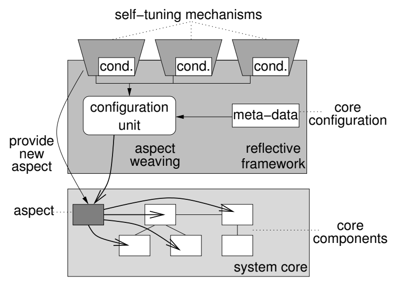

Fig. 2. Self-tuning using aspect weaving

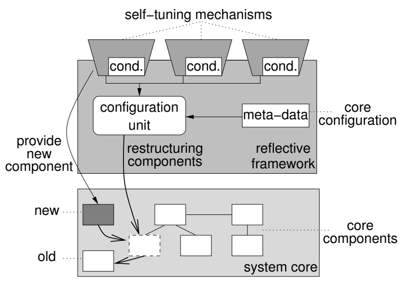

Fig. 3. Self-tuning using component composition

Component Techniques. Configuration is also feasible by the composition of components. A so-called *configuration unit* recomposes the core components. In other words, it exchanges components, applies new ones and removes existing ones. Figure 3 shows the replacement of an existing component (the white box at the bottom left) by a new one (gray box). One can elegantly encapsulate behavioral strategies into components. For instance, an overlay network peer can pursue different caching policies, e.g., neighbors cache data frequently accessed, more remote peers cache data, etc. A peer can change its caching policy by exchanging the respective component by another one with the same interface.

Architectural Reflection. Architectural reflection uses design information and evolutionary rules to evolve the structure and behavior of software systems [6]. Our idea is to combine this approach with our self-tuning mechanisms. Each self-tuning mechanism does not only provide consistency conditions but also evolutionary rules. Note that evolutionary rules are different from consistency conditions. Consistency conditions prevent from conflicts between self-tuning mechanisms. Evolutionary rules in turn describe *how* to adapt the structure and the behavior of the peers. We hypothesize that overlay network-specific evolutionary rules can describe the specific structure, behavior and characteristics of overlay networks in a more precise way than generic rules [6].
Figure 4 depicts overlay networks enhanced by architectural reflection. An application programmer can obtain information on the overlay network-internal state and structure using the meta-interface. Meta-objects bear this information. The programmer defines evolutionary rules inside a self-tuning mechanism to specify which characteristic is observed and how to adapt the corresponding parameter. Rules have the form (*if*{<*condition*>} *then* {<*adaptation*>}). The configuration unit processes the rules and observes the characteristics specified by the *if*-block. If the conditions become true it executes the *then*-clause. The *then*-clause modifies the reified meta-objects. Finally, the configuration unit reflects the modifications into the base-level.

Currently, we do not know which method is superior in our specific context. AOP and dynamic weaving are able to modify the system core at expression and statement level. The advantage is that one can implement the self-tuning mechanisms (which are crosscutting concerns) in modular units. The disadvantage is that the weaving process becomes complex. This might result in decreased performance and a higher resource consumption. However, an advantage is that we can reuse aspects in overlay networks which have been implemented for other domains. There are some good candidates from the middleware domain, namely synchronization, logging, authorization, load balancing, and encryption.

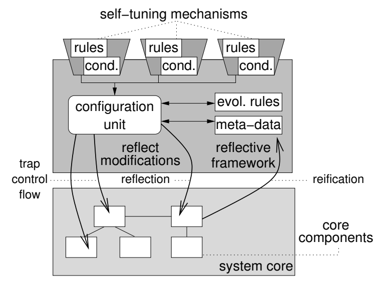

Fig. 4. Self-tuning using architectural reflection

The advantage of modern component models is that components are easy to compose. However, they can interact with the system core on method-, class- and object level only, in contrast to statement-level with AOP. Exceptions are the component languages FEATUREC++ [1] and CAESARJ [16], but both are currently still immature. Because of their crosscutting nature, not all self-tuning mechanisms can be implemented in a highly modular way. This may lead to implementation-level interference. Further, not all component models provide dynamic composition. However, modern component models are base technology to build PLA.

Similarly to the dynamic weaving approach, architectural reflection is extremely flexible. The application programmer decides at expression/statement level where the control flow is trapped, which base data is reified, and which evolutionary rules specify the new system behavior/structure. The disadvantage are higher memory consumption (to store design information) and weaker performance, since design information is processed, reified and reflected at runtime. An open question is how to specify and check such evolutionary rules in the context of overlay networks.

7 Preliminary Implementation Results

This section present our first experiences in implementing self-tuning mechanisms. Our preliminary approach combines AOP and Mixin Layers.

Our first attempts to implement self-tuning were contact caching, the reputation mechanism and load balancing. During the implementation of these mechanisms we observed that the dissemination of meta-data is fundamental to most self-tuning mechanisms. Examples for meta-data are information about the current load, remote contacts, feedback objects of our reputation mechanism, RSA-keys, and application specific information, e.g., synchronization signals in parallel processing applications (see [5] for a discussion). Due to the absence of a central coordinator the meta-data must be propagated in a decentralized swarm-like manner. We see several strategies to cache incoming meta-data, e.g., FIFO, Random, LRU, etc. (*caching strategies*), as well as to propagate it to the 'right' direction, e.g., Greedy Forwarding, Kleinberg distribution, CAN Multicast, etc. (*dissemination strategies*).

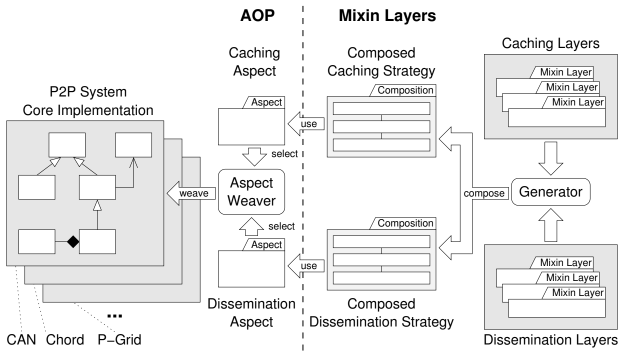

Fig. 5. Generating and applying strategies using Mixin Layers and AOP.

Furthermore, we have observed that strategies often differ only in details, e.g., a threshold or some additional constraints. Out of these observations and the considerations of Section 6 we investigated in an combined, component-based and aspect-oriented architecture. We utilize AOP and Mixin Layers that enable us to implement a broad range of caching and dissemination strategies on top of an overlay network. These strategies are essential to implement higher-level self-tuning mechanisms. Figure 5 gives an overview of our approach. The following paragraphs discuss this approach and the roles of AOP and Mixin Layers.

AOP. Mainly, we have used AOP (AspectJ) to separate the code for the meta-data caching and dissemination strategies from the core implementation of the overlay network. The left side in Figure 5 depicts a schematic design of dissemination and caching strategies implemented as aspects. It can be seen that the aspects are separated clearly. This allows us to design a generalized protocol implementation that does not depend on the type of the overlay network and increases the reusability and the ability of plugging strategies.

Mixin Layers. During the design phase we have observed that we need often only small modifications of the caching and dissemination strategies to specify different protocol characteristics. For example, from the algorithmic point of view the caching strategy random differs from FIFO only in the order the items are removed. Furthermore, some features are common for many strategies, e.g., to keep track of the peer a certain item was already sent to. Implementing a new strategy in form of a separate aspect for each variant would be inefficiently and errorprone, because approved code is not reused. Therefore, we have decided to combine the approach of aspect-oriented strategies with a Mixin Layer-based implementation.
We have used Mixin Layers to implement variants of the different caching and dissemination strategies and the *AHEAD Tool Suite* [3] to automatize the configuration and generation process. Doing, so we had to decompose the mechanisms for caching and dissemination into fine-grained layers. Based on this design we are able to compose these layers to generate different strategy variants. AOP is then used to apply the configured strategies to the core of the overlay network

peers. Figure 5 depicts the schematic overview of configuring and applying strategies to the peers: At first, a generator (here the *AHEAD Composer*) composes the strategies according to a (user) specification. Then, the composed strategies are applied to the peer core using aspects and the *AspectJ Weaver*. Using Mixin Layers, we do not need multiple aspects. Instead, we need only two of them: one for applying the caching strategies and the other for applying the dissemination strategies. Figure 6 depicts the stack of Mixin Layers implemented for random dissemination and FIFO caching. The enclosing grey boxes are the Mixin Layers. The included rounded boxes are the inner Mixin classes. The figure depicts only the inheritance relations between the inner classes. Other relations (e.g. associations) as well as references to external classes are omitted. The *Base* layer includes the needed data structures: *MetaData*, to provide a base for application specific data, *MetaDataCache* for storing the cached meta-data, *MetaDataList* to attach meta-data to messages, as well as the strategy base classes (interfaces) *CachingStrategy* and *DisseminationStrategy*. Tangible strategies (in our example random dissemination and FIFO caching) implement these interfaces to provide the desired algorithm. The layer *TimeStampedMetaData* refines the *MetaData* base by adding time stamps to each meta-data object. The layer *NoRedundantEntries* prevents the meta-data cache and the messages from storing equal meta-data objects. *ExperimentLogging* creates the logging dump which we used to analyze our experiments.

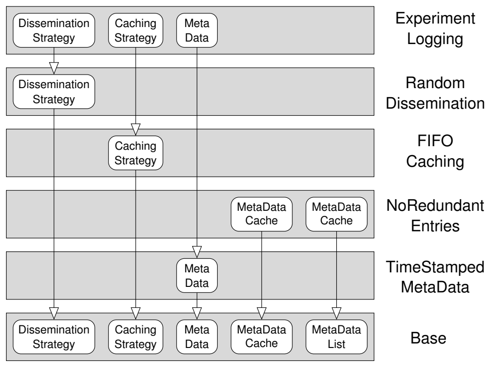

Fig. 6. Stack of Mixin Layers for FIFO Caching and Random Dissemination.

The advantage of using Mixin Layers is that we can easily derive and combine new variants of strategies. To introduce a new caching strategy we have to implement the interface *CachingStrategy*. Doing so, we can remove the currently used strategy and apply the new one. In this way we can easily combine caching strategies with dissemination strategies. Different application-specific self-tuning mechanisms require different combinations of caching and dissemination strategies [5]. A further advantage is that we are able to reuse the base data structures as well as existing strategy implementations. Imagine a proximity FIFO caching strategy which combines FIFO caching with the contact information of the peer. This might be useful in order to limit the range of meta-data to the contacts of a peer. If a cache entry was generated by a immediate contact, the caching strategy is FIFO. If not, the cache entries are dropped. The proximity FIFO caching strategy reuses the overall code of the simple FIFO strategy.

Discussion. This implementation concept can be used to implement more complex self-tuning mechanisms (we argue). In current we have not considered semantic interference and dynamic plugging. We propose the use of an extended version of the AHEAD Design Rule Checks to implement the consistency conditions. To allow dynamic plugging the aspects for applying the self-tuning components, e.g., Mixin Layer stacks, must be dynamically weaveable. For that aspect frameworks like *PROSE* [17] or *AspectWerkz* [27] can be used. The consistency conditions can be processed at runtime by parsing the corresponding files (*.drc-files*). In summary the combination of AOP and Mixin Layers is one appropriate approach to implement modular self-tuning. In future work we will additionally investigate in other component-techniques and architectural reflection as well as the seamless integration of design rules and the overall reflective architecture.

8 Organic Overlay Networks and Emergent Behavior

Beside the modelling of self-organizing overlay networks as well as design and implementation issues, we investigate in the potential organic character of rule- and state-based self-organizing overlay networks. Our model of self-organizing overlay networks has all characteristics of organic *vivi-systems* [12]: (1) no central coordination instance, (2) individual peers are autonomous, (3) peers are interconnected massively, and (4) causal dependencies of peers and their behavior may be non-linear. This observation leads us to the question if self-organizing overlay networks behave organic in the sense of an emergent behavior [28]. Naturally, we are interested how to model the behavior of peers as well as their interactions in general to achieve an emergent order. This means that the global behavior of the overlay network shows an ordered pattern that cannot be easily derived from the local behavior (described by states or rules) of the peers. To find indices for the emergence of order we are experimenting with swarm-algorithms. In particular we try to adopt ideas of flocking models [21] and social interaction models [2] to implement self-tuning mechanisms. First results show that the a organic behavior can be achieved by simulating swarms a the top of a CAN. Currently, we want to apply these results to a simple load balancing mechanism. Thereby, the load balancing algorithm is not described as a global entity or policy (as in [11]). Instead it is distributed over the individual peers in form of behavioral rules. Therefore, the desired global behavior, a balanced load, emerges from the behavior of the autonomous peers. One main goal is to find and identify so called phase transitions. Phase transitions are spontaneous changes of the behavior triggered by minimal parameter adjustments. In first experiments we have found indications for such phase transitions in a culture model [2] on top of a CAN. We are interested in which parameters of an overlay network can trigger such phase transitions and how to adjust the parameters to shift for instance the load distribution of the overlay network to a ordered state (a balanced load). From these insights we want to abstract, to build more efficient, complex self-tuning mechanisms, e.g. our reputation mechanism. Finally, we will investigate in the parallel execution of different self-tuning mechanisms and the impact on the emergent global behavior, phase transitions and robustness against failures.

9 Conclusions

Overlay networks are useful for a broad range of applications and must cope with different and changing environments. This calls for self-organization. This article provides an overview of the current status of our ongoing project that targets at self-organizing overlay networks. We have described the design decisions and problems that we currently face. For instance, self-tuning mechanisms may interfere with each other and with the system core. Such interference can occur at implementation and semantic level. We have discussed different methods to deal with this problem in the context of overlay networks. Furthermore, we have compared different techniques to implement self-tuning mechanisms. Thereupon, we have presented a first approach based on AOP and Mixin Layers. As part of our future directions, we have presented our ongoing work on organic overlay networks and emergent behavior. A set of current and ongoing experiments with swarm-algorithms based on overlay networks are supposed to reveal insight into organic overlay networks. Thereupon, we want to build several new self-tuning mechanisms that trigger an emergent order.

References

1. S. Apel et al. FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++. Technical report, Department of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2005.
2. R. Axelrod. The Dissemination of Culture: A Model with Local Convergence and Global Polarization. *Journal of Conflict Resolution*, 41, 1997.
3. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.
4. K. Böhm and E. Buchmann. FairNet – How to Counter Free Riding in Peer-to-Peer Data Structures. In *Proc. of COOPIS*, 2004.
5. E. Buchmann and S. Apel. Piggyback Meta-Data Propagation in Distributed Hash Tables. In *Proc. of WEBIST*, Miami, Florida, May 2005.
6. W. Cazzola et al. Rule-Based Strategic Reflection: Observing and Modifying Behavior at the Architectural Level. In *Proc. of ASE*, 1999.
7. A. Datta, M. Hauswirth, and K. Aberer. Beyond "Web of Trust": Enabling P2P E-Commerce. In *Proc. of the IEEE Conf. on E-Commerce*, 2003.
8. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
9. J. Dowling and V. Cahill. The K-Component Architecture Meta-model for Self-Adaptive Software. In *Proc. of Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*, 2001.
10. S. D. Gribble et al. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Computer Networks*, 35(4), 2001.
11. D. Hughes, G. Coulson, and I. Warren. A Framework for Developing Reflective and Dynamic P2P Networks (RaDP2P). In *Proc. of P2P Computing*, 2004.
12. K. Kelly. *Out of Control: The New Biology of Machines, Social Systems, and the Economic World.* Basic Books, 1997.
13. G. Kiczales et al. Aspect-Oriented Programming. In *Proc. of ECOOP*, 1997.
14. G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol.* MIT Press, 1991.

15. P. Maes and D. Nardi, editors. *Meta-Level Architecture and Reflection*. North-Holland, 1988.
16. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT*, 2004.
17. A. Popovici, G. Alonso, and T. Gross. Just in Time Aspects: Efficient Dynamic Weaving for Java. In *Proc. of AOSD*, 2003.
18. V. Ramasubramanian and E. G. Sirer. The Design and Implementation of a Next Generation Name Service for the Internet. In *Proc. of ACM SIGCOMM*, 2004.
19. A. Rao et al. Load Balancing in Structured P2P Systems. In *Workshop on Peer-to-Peer Systems*, 2003.
20. S. Ratnasamy et al. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM*, 2001.
21. C. W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics*, 21(4), 1987.
22. M. Romn, F. Kon, and R. Campbell. Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online*, 2(5), 2001.
23. A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of Middleware*, 2001.
24. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM TOSEM*, 11(2), 2002.
25. B. C. Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, MIT Laboratory for Computer Science, 1982.
26. I. Stoica et al. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, 2001.
27. A. Vasseur. Dynamic AOP and Runtime Weaving for Java - How does AspectWerkz Address it? In *Workshop on Dynamic AOP*, 2004.
28. E. Yates, editor. *Self-Organizing Systems: The Emergence of Order*. Plenum, New York, 1987.