OverArch: A common architecture for structured and unstructured overlay networks

Ingmar Baumgart, Bernhard Heep, Christian Hübsch
Institute of Telematics
Karlsruhe Institute of Technology (KIT)
Zirkel 2, D–76131 Karlsruhe, Germany
Email: {baumgart, heep, huebsch}@kit.edu

Amos Brocco
Department of Innovative Technologies,
University of Applied Science of
Southern Switzerland (SUPSI)
Email: amos.brocco@supsi.ch

***Abstract*—There exists a variety of different peer-to-peer (P2P) protocols to support a wide range of distributed services, such as content distribution or data storage. In order to promote interoperability and facilitate the development of new P2P applications, common application programming interfaces (APIs) have been proposed. Unfortunately, most of these interfaces have stagnated, and fail to meet present research or business requirements. In this regard, this paper presents a novel common architecture and API which combines structured and unstructured overlay networks and strives to overcome the limitations of previous architectures. Our work defines a set of generalized components that are common in today's P2P systems, and provides a clean interface that facilitates the rapid development of new P2P applications and services. We validate the proposed architecture by presenting a concrete implementation including a broad range of protocols within the P2P simulator OverSim.**

I. Introduction

Even in the early days of peer-to-peer (P2P) systems, researchers and developers realized that a common application programming interface (API) was necessary to foster the creation of innovative solutions while simplifying comparison between systems developed by independent parties.

An important step in this direction was made with the definition of a *Common API for structured peer-to-peer overlays* [1] (in the following called *Dabek API*), which laid down a path toward interoperable applications, protocols and services. That effort focused on a layered architecture composed of basic building blocks: each block implements a clear semantic and supports high-level abstractions to facilitate the development of applications running on structured peer-to-peer overlays. The Dabek API has been adopted by several projects (e.g. [2]) in the past. Currently there is also a strong interest from the industry to have such well-defined common building blocks, which helps to rapidly develop novel P2P applications. The *IETF* currently supports this demand and works on the standardization of a modular P2P protocol [3].

Over time, research has led to the development of improved techniques for solving common problems such as routing and content delivery. In the light of this evolution, the aforementioned protocol architecture exhibits many shortcomings that undermine applications' portability and interoperability, by forcing developers to cut corners around known API limitations. Through our experience with the implementation of several P2P protocols on the OverSim framework [4], we identified several directions for improving the Dabek API. Accordingly, in this paper we present a novel architecture named *OverArch* and a corresponding programming interface focusing on a broad range of P2P overlay networks. Our solution strives to overcome the limitations of previous proposals by presenting a unified interface for both structured and unstructured topologies. The proposed approach is based on a modular architecture which exposes several high-level primitives to provide services such as *Key Based Routing (KBR)*, *Key-independent Message Dissemination (KIMD)*, *Distributed Data Storage (DDS)*, and *Application Layer Multicast (ALM)*. Each component is accessible through a clean interface which hides the details of the underlying implementation. In addition, we propose to modularize the KBR component itself and identify reusable building blocks like *routing* or *timeout handling*.

The remainder of this paper is organized as follows: in the following section the shortcomings of the Dabek API will be discussed. In Section III *OverArch* is presented, followed by the interface description in Section IV and the modular KBR component in Section V; a validation of our approach is presented in Section VI. Section VII gives an insight on other efforts toward the definition of a common API for P2P systems; finally, Section VIII summarizes our work.

II. Shortcomings of the Dabek Common API

The Dabek API [1] was an important first step towards a generic and flexible P2P architecture. Especially the concept of separation between routing (*KBR*) and storage (*DHT*) had a great influence on further P2P systems. On the other hand the scope of the API is *limited to structured P2P systems*, which disregards the large group of unstructured proposals. Another shortcoming is the *strict layered* architecture, that is too inflexible for current complex P2P systems.

A major concern is the imprecise definition of *overlay neighbors*. There is no clear separation between the used terms *neighbor set*, *replica set* and *r-roots*. For an efficient and secure distributed storage service, it is e.g. desirable to have a symmetric neighborhood relationship for storing replicas, which is not covered by the *r-root* definition of the Dabek API.

Additionally there are many details, which turn the Dabek API unsuitable for specific protocols or applications: e.g. the proposed *range()* method cannot be realized with Kademlia's [5] XOR-based distance metric. Another limitation is

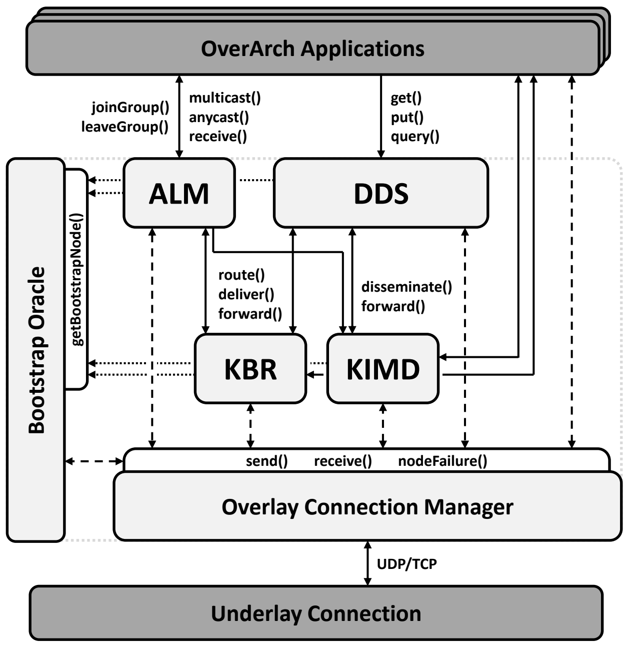

Fig. 1: Modular overlay architecture *OverArch*: Building blocks, service components, and interfaces

the *focus on recursive routing* (e.g. there is no method for iterative lookups). Finally, the Dabek API details just the KBR interface, providing only stubs for the remaining interfaces, like *distributed storage* or *application layer multicast*.

III. A modular Architecture for Overlay Systems

For defining an architecture for P2P systems that covers the functionality of structured and unstructured overlay protocols as well as overlay services like ALM and distributed data storage, we identified components of today's common P2P systems. These components can be considered as exchangeable building blocks or "black boxes" providing well defined APIs to other components and user applications. By assembling a subset of these components depending on the requirements of a new P2P application this modular approach supports and eases its development. An example for this could be a *Content Distribution Network (CDN)* [6] which allows for an efficient distribution of (large) data objects. The CDN application might use the DDS component to locate other users that hold missing parts of requested data objects.

Fig. 1 gives an overview of *OverArch*. The main building blocks are the *Underlay Connection* which provides basic connectivity through TCP/UDP, the *Overlay Connection Manager* holding information about all overlay nodes known to the local node, the *Bootstrap Oracle* for encapsulating the utilized bootstrapping routine, and the service components KBR, KIMD, ALM, and DDS.

An advantage of this modular architecture is the fact that it can be either considered as a layered or a component based architecture, i.e. connections between components are not limited by a strict layer architecture. This enables to have several applications, which make use of the same KBR component or to have a common Bootstrap Oracle component which is connected to several components like KBR or DDS. P2P applications based on this architecture are free to choose concrete protocols that realize the service components according to its requirements.

Before detailing the identified components and their APIs we define the most important terms and data structures:

- ***NodeIds / Keys***: *NodeIds* and *keys* are elements of the same *id space* $\mathbb{Z}_{2^m}$ with a *distance metric* assigned to in order to determine distances between its elements. Usual choices for m are 128 or 160, which results in a low risk for collisions with randomly selected nodeIds. While *nodeIds* are assigned to every node in the overlay network as unique identifiers, keys generally represent objects.
- ***Transport Address***: The *transport address* of a node X contains all necessary data (e.g. IP address and UDP port) to address the node in the underlay network.
- ***Node Handle***: The *node handle* of an overlay instance X comprises a *transport address* and an additional *nodeId* corresponding to the specified node. Node handles are used as entries in overlay routing tables representing neighbors in the overlay network.
- ***Overlay Neighbor***: An overlay node Y is neighbor of a node X, if there is a direct connection in the overlay topology from X to Y, i.e. Y is an entry of X's overlay routing table.

IV. Components and Interfaces

In the following, we introduce the identified main building blocks and service components and describe their interfaces.

A. Overlay Connection Manager

Basically, the Overlay Connection Manager maintains a list of all overlay nodes known to the local node including its neighbors in the overlay. This ensures consistency as entries in the service components' routing tables are always pointers into this list. Furthermore, this list holds common information about the overlay nodes. This comprises the nodes' node handles, their uptime, their physical network distance given by e.g. the *Round-Trip-Time (RTT)*, detected node failures, and information for NAT traversal. Node failures might be detected by the service components which provide this information to the Connection Manager using the *nodeFailure()* method.

By using a transport address as destination address a component is able to address an overlay instance on a specific overlay node. Moreover, since all messages that are sent between overlay nodes pass the Overlay Connection Manager, components on an overlay node are addressable: the Connection Manager acts as a multiplexer by setting the source component field (*srcComp*) of outgoing messages and using the destination component fields (*destComp*) of all incoming overlay messages to decide to which component the message is forwarded.

B. Bootstrap Oracle

The Bootstrap Oracle provides the transport addresses of potential bootstrap nodes to all service components. For this, the component that needs to know a bootstrap node calls the *getBootstrapNode()* method. Since the mechanism used to retrieve these transport addresses is hidden, the Oracle can abstract various possible bootstrap mechanisms like probing or web-based bootstrapping.

C. Key-based Routing

The *Key-based Routing (KBR)* component provides efficient *message forwarding* to keys over a *structured* overlay topology (e.g. ([7], [8], [5], [9]).

The KBR component provides the *route()* method to forward a message *msg* to the node, which nodeId is closest to the *destination key* according to the distance metric. For this purpose, each node maintains a *routing table* that holds entries of overlay neighbors. Each *routing table entry* consists of at least one *node handle*.

In [1] the KBR service is used as building block for a distributed storage system called *distributed hash table (DHT)*. In order to improve the availability of the stored data items such a storage system stores several replicas of all data items on different overlay nodes. The closest nodes to the keys of data items are particularly suitable for storing these replicas. To ease the implementation of a robust DHT, we introduce the concept of *siblings* described later.

In the following we give an overview of our KBR interface and compare it to the Dabek KBR interface [1].

- ***route**(key, destComp, msg, [srcRoute], routingMode)*
  The method *route()* forwards the message *msg* via several routing hops (if required) to the node which is closest to the destination *key*. The parameter *destComp* is used for addressing the destination component (e.g. DDS or ALM) in case that several components are connected to the KBR component. In contrast to our component-based architecture the Dabek API employs a strict layer-based approach and supports only a single service above the KBR layer, which we consider too inflexible.
  Another aspect neglected by the Dabek API is an option to specify several next hop candidates or even a complete source route. This is supported by our API with the *srcRoute* parameter, containing a (optional) list of transport addresses. Finally the *routingMode* parameter offers a choice between different routing modes (e.g. *semi-recursive*, *full-recursive* or *iterative*).
- $\langle$ *result, error* $\rangle \leftarrow$ ***isSiblingFor**(node, key, s)*
  The method *isSiblingFor()* is used to verify, if a given *node* is a so called *sibling* for the given *key*. *Siblings* are the s closest nodes to a key, determined by the metric of the id space. The set of nodes, which are responsible for a key key, is named $\mathcal{S}^s_{key}$ in the following (with $|\mathcal{S}^s_{key}| = s$). The method *isSiblingFor()* replaces the method *range()* of the Dabek API, since the functionality of *range* as described in [1] can not be provided with some structured overlay protocols (e.g. Kademlia [5]).
  In contrast to the so called *r-roots* defined in the Dabek API, our *siblings* define a *symmetric* neighbor relationship. This means, that a node which belongs to the closest *s* nodes to a given *key* knows all other (*s - 1*) closest nodes for the key by querying its local routing table. The sibling neighborhood is an important concept for providing a secure and efficient distributed storage service (see section IV-E).
  The Boolean return value *result* is true if the given *node* is a sibling for the given *key*. If the local node cannot unambiguously determine if *node* is a sibling, this is indicated by the Boolean return value *error*.
- *siblings[]* $\leftarrow$ ***lookup**(key, s, routingMode)*
  The method *lookup()* is used to determine the *s siblings* for the given *key* using several routing hops if required. The parameter *routingMode* specifies the applied routing mode. The method returns a list of *node handles* sorted by the distance to *key*.
  The Dabek API does not offer a similar method. We consider *lookup()* an important method when using iterative routing modes, since it enables to efficiently get a list of all siblings for a key. This list can be used e.g. by the DDS component to directly contact all replica nodes for a key, which is more secure then routing a single message through the main replica node.
- *nodes[]* $\leftarrow$ ***getClosestNodes**(key, r, s)*
  The method *getClosestNodes()* returns a list of node handles of the *r* closest known nodes for destination *key* from the *local routing table*. If the method is called on node X with $X \in \mathcal{S}^s_{key}$, the *s* siblings for *key* are returned. For $s = r = 0$, this method returns a complete dump of the local routing table.
  The *getClosestNodes()* method replaces the three methods *local_lookup()*, *neighborSet()* and *replicaSet()* of the Dabek API with a single method.
- *success* $\leftarrow$ ***join**([nodeId])*
  By calling *join()* the local node joins the overlay with the optionally given *nodeId*. The specification of a dedicated nodeId is e.g. needed when using cryptographically secured nodeIds (e.g. [10]) or when using *topology-based nodeId assignment*. The Dabek API in contrast only offers that the nodeId is generated by the overlay itself.
- *s* $\leftarrow$ ***getMaxNumSiblings**()*
  The method *getMaxNumSiblings()* returns the maximum number of available siblings. This depends on the employed routing protocol and is usually parameterizable by increasing the routing table size.
- $\rightarrow$ ***deliver**(key, srcComp, srcNode, msg)*
  This upcall delivers the message *msg* at the addressed destination component of the node responsible for *key*. Additional information include the originator of the message (*srcNode*) and the source component (*srcComp*).
- $\rightarrow$ ***forward**(*key, destComp, srcComp, srcNode, *msg)*
  This upcall is used at every node on the routing path to the destination *key* to inspect and (optionally) modify a passing message *msg* by all attached components. *forward()* can also change the destination *key*.
- $\rightarrow$ ***update**(node, joined)*
  By calling the *update()* method in all connected components, the KBR component signals a change in the sibling neighborhood of the local node. If the parameter *joined* has the value *true*, the node *node* has joined the sibling neighborhood. Otherwise the node has left the sibling neighborhood. This notification is e.g. used by the DDS component to trigger the transfer of replicas to new siblings.

D. Key-independent Messages Dissemination

The *Key-independent Message Dissemination (KIMD)* component provides *message dissemination* to other overlay nodes over an *unstructured* overlay topology ([11], [12]). For this, each overlay node connects to other nodes—its neighbors—chosen by protocol-specific criteria (e.g. network load, number of neighbors). Dissemination is either done by flooding the overlay network or by performing (biased-)random walks.

In the following we give an overview of our KIMD interface:

- ***disseminate****(msg, destComp, fanOut, htl)*
  A node calls the *disseminate* method if it wants to send a message *msg* to all other nodes in the overlay network. The *fanOut* parameter can be considered as the flooding multiplicator of the procedure, i.e. it determines the number of neighbors the message is forwarded to at each node. If set to 1, a random walk is performed. The *htl* parameter (hops to live) determines the maximum number of hops for a single message. A KIMD-message's header comprises a hop count field initialized by the value of *htl*. At each node the message passes the hop count field is decreased. If the hop count reaches zero the KIMD component drops the message.
- $\rightarrow$ ***forward****(*msg, destComp, srcComp, originator lastHop, *fanOut, hopCount, *htl)*
  This upcall method is triggered when a node receives a message so that the called component (or application) can interpret the message and react in an appropriate way. For this, the transport address of the message's *originator*, the message's *lastHop*, and the configured *fanOut* parameter are passed to the component. Then, the component decides whether the message is dropped or the dissemination continues. If so, the component may optionally modify *msg*, *fanOut* and *htl*.
- *neighbors[]* $\leftarrow$ ***getNeighbors****()*
  addNeighbor*(neighbor)*
  removeNeighbor*(neighbor)*
  These methods give access to the managed overlay neighborhood by the KIMD component. It is possible to dump the complete neighborhood (*getNeighbors()*), add a single neighbor (*addNeighbor()*), and remove specific nodes from the overlay's neighborhood (*removeNeighbor()*).

Based on the KIMD component a *gossiping* service can be realized by calling the *dissemination()* method periodically.

E. Distributed Data Storage

The *Distributed Data Storage (DDS)* component stores data objects, which are identified by a *key* and additional *meta information*, like e. g. an index. A DDS service can be realized either on top of a KBR or a KIMD component. If using a KBR component, this service is commonly known as *distributed hash table (DHT)*, which allows to efficiently retrieve a data record according to its key. A DDS service on top of a KIMD component instead uses flooding or random walk to find stored data records according to some filtering criteria.

The DDS component provides the following interface:

- ***put****(key, value, ttl, [metainfo], [filter])*
  The method *put()* stores a data object with a given *key* and *value*. The parameter *ttl* specifies the time-to-live in seconds, after which a stored object gets deleted. An object may have additional *metainfo* like an index, which can be used to store several objects with the same key. If there are already stored data objects with the given key and their metainfo matches the given *filter*, these objects get replaced by the new object. If *value* is empty, a matching object gets deleted.
- $\langle$*value[], metainfo[]*$\rangle$ $\leftarrow$ ***get****(key, [filter], [node])*
  The *get()* method retrieves data objects, which match the given *key* and an optional *filter* on metainfo. It is also possible to query only data objects, which are stored on a specific *node*. The method returns a list containing the *value* and corresponding *metainfo* of all found objects.
- *nodes[]* $\leftarrow$ ***query****(key, [filter])*
  The *query()* method is similar to the *get()* method, but returns only a list of *nodes* which store matching data objects, instead of returning the data objects themselves. This is useful if the DDS operates over an unstructured overlay network, in which a query gets flooded and reaches several nodes with matching data objects. If data objects are large, it is more efficient in this case to return only a pointer to these nodes and get the actual object later by using *get()*.

F. Application Layer Multicast

The *Application Layer Multicast (ALM)* component provides multicast and anycast services on application layer. Either ALM services build up their own unstructured overlay [13] to send messages to groups of nodes or they make use of a KBR (e.g. [14]) or a KIMD component (e.g. [15]). The ALM component provides the following interface:

- *success* $\leftarrow$ ***joinGroup****(group)*
  A node wishing to join a specific ALM group calls *joinGroup()*, providing the specific group's ID *group*. A group always comprises all nodes that have previously joined it. A group's ID is represented by an unique key, e. g. being a group's name as a string or a hash value from that string. In case a node joins an empty group (comprising no further nodes), it is considered this group's initiator.
- *success* $\leftarrow$ ***leaveGroup****(group)*
  Leaving a group results in the node being removed from the respective ALM structure. By calling *leaveGroup()*, a node stops receiving data from that group instantly.
- ***multicast****(msg, group)*
  To send a message to a specific group, a node provides the message and the group ID to the method *multicast()*. Multicasting a message to a group results in the message being delivered to every node being part of the group.
- ***anycast****(msg, group)*
  Besides multicasting data, a node can may also send a message to a single arbitrary node in the group by calling *anycast()*. This functionality has to be provided by the ALM protocol for the specific use case.
- $\rightarrow$ ***receive****(msg, group)*
  A node having joined a group will receive any message that is sent to this group via the *receive()* callback.

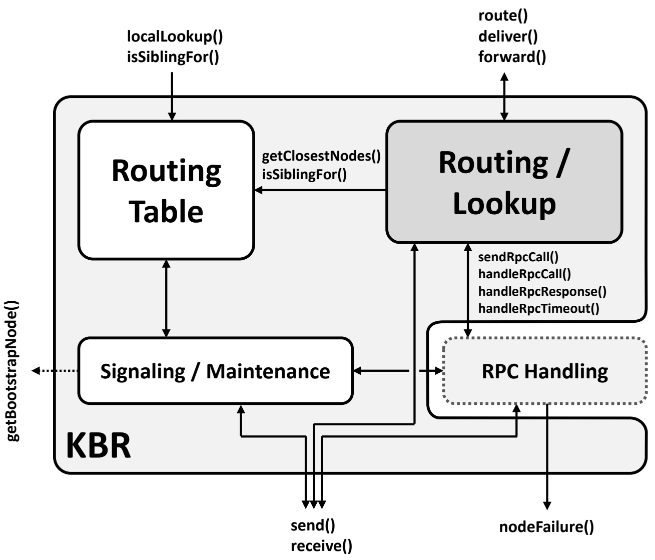

Fig. 2: Internal architecture and interfaces of the modular KBR component

V. Modular KBR Architecture

In Sec. IV-C we described the KBR API that is offered by the KBR component to other components and applications. In this Section we introduce our approach for a modular architecture of a generic KBR component, including a specification of its sub-modules and internal interfaces. The insights expressed with this approach were gained during the implementation of several KBR protocols for the *OverSim* simulation framework, i.e. the KBR functionality implemented in *OverSim* is based on the proposed architecture.

As shown in Fig. 2 the modular KBR component comprises four modules:

- ***RPC Handling***: The *Remote Procedure Call Handling* module handles all issues regarding messages that are sent (*calls*) for which *response* messages are expected. These issues include message context maintenance, timeout handling and message retransmission (if required). Node failures detected on the basis of missing response messages are notified to the *Overlay Connection Manager* by calling *nodeFailure()*. RPC Handling is not necessarily part of the KBR component, i.e. it can be considered as a core mechanism of *OverArch* usable by all other service components, the *Bootstrap Oracle*, and the applications. In *OverSim* the RPC module is implemented in an abstract base class all components can derive from.
- ***Routing and Lookup***: This main module of the KBR component organizes all routing and lookup procedures the local node is involved in. When an application initiates a recursive routing procedure (by calling *route()*) or when the node receives a message that is addressed to a given destination *key*, this module first asks the routing table if the local node is responsible for the destination key (by calling *isSiblingFor(thisNode,key,s)*)). If so, the message is passed to the component addressed in the message (by calling *deliver()*). If not, the module asks the routing table for potential next hop candidates to forward the message to (by calling *getClosestNodes(key,r,s)*).
  For iterative lookup procedures the module sends out (one or more) *getClosestNodesCalls* (using the *RPC Handling* module) to nodes from a vector v. This vector contains nodes sorted by the distance to the lookup *key* and is initialized by a local call of *getClosestNodes(key,r,s)*. The Lookup/Routing modules at the receiver nodes send back *getClosestNodesResponses* that contain lists of close nodes to *key* (compiled by *getClosestNodes()*). These nodes are merged into v and new *getClosestNodesCalls* are sent out to the first nodes of v that have not contacted yet. This procedure is repeated until the responsible node for *key* is identified.
- ***Routing Table***: This module holds the overlay routing table structure, i.e. it represents the node's neighborhood in the overlay network. As it is an abstract module, its concrete implementation is determined by the utilized overlay protocol. However, to maintain consistency all entries must be pointers into the overlay node list held by the *Overlay Connection Manager* component. The *Routing Table* module offers a protocol-independent interface to external components and other KBR-internal modules. The interface comprises the methods ***isSiblingFor()*** and ***getClosestNodes()*** as described in Sect. IV-C.
- ***Signaling and Maintenance***: The protocol-dependent Signaling and Maintenance module is responsible for maintaining the routing table. For this, it performs reactive or periodic tasks, i.e. sends signaling messages to other overlay nodes or routes signaling messages to keys that are determined by the utilized overlay protocol. For doing this, the RPC module can be used if required.

By decoupling the routing table and the routing/lookup mechanisms this architecture allows for the usage of different recursive and iterative routing/lookup modes independent of the utilized overlay protocol. Another benefit of this approach is the fact that for implementing new protocols the main routing/lookup module and the RPC handling can be reused. Only the protocol-dependent routing table and the maintenance module have to be implemented.

VI. Validation

To validate our architecture, we successfully implemented several structured (e.g. *Chord* [7], *Kademlia* [5], *Pastry* [8], *Bamboo* [9], *Broose* [16], *Koorde* [17]) and unstructured overlay protocols (e.g. *GIA* [12], *NICE* [13]) as well as a common DDS component in our simulation framework *OverSim* [4]. It has been shown that all of these protocols fit very well into the OverArch architecture (only Chord showed a small limitation, since it does not provide symmetrical siblings).

Additionally, we implemented an ALM-based application based on the service provided by the *Scribe* [14] ALM component. Scribe in turn uses Pastry as KBR component. We also successfully implemented more complex P2P applications like the *P2PNS* [18] distributed name service or a distributed service for massively multiplayer online gaming.

In addition to internal usage, the API is also exposed by OverSim with an XML-RPC interface to external applications and can be used similar to the XML-RPC interface of

OpenDHT [19]. All these implementations are published as open-source project (*http://www.oversim.org/*).

VII. Related Work

Beside the already described Dabek API [1] for structured P2P networks, there are several other proposals for common application programming interfaces, which restrict themselves to specific instances of P2P services, such as DHTs [19], ALM [20], publish subscribe [21][22], or focus on cross-layer concerns to fulfill the needs of mobile networks [23]. To our knowledge there is no common API which has been shown to be suitable for both, structured and unstructured overlay networks.

VIII. Conclusion

Standardized application programming interfaces (APIs) are meant to promote consistency, reduce programmer's learning curve, and ease porting of applications. In the area of distributed systems a well known example is the *Dabek API* [1], which dates back to 2003. Unfortunately, today's protocols and services have exposed several shortcomings of this API, such as the missing support for unstructured protocols, an incomplete API for distributed storage or ALM and finally an inflexible layer-based approach. These deficiencies greatly narrow the scope of the Dabek API, and substantially hinder the development of truly interoperable distributed applications.

In this paper we presented our effort to define a new standard for the rapid development and prototyping of novel P2P applications, which resulted in a modular and flexible architecture made of exchangeable blocks that implement comm'on functionalities of today's P2P systems. In particular, we detailed the most important terms and data structures used by the components and their APIs, show the feasibility to modularize the KBR component in smaller building blocks and finally validated our work by presenting the implementation of several structured and unstructured overlay protocols, as well as complex P2P applications in our OverSim framework.

In the future we plan to include other overlay-based services (e.g. content distribution networks like *BitTorrent*) in the *OverArch* architecture and define a common format for overlay messages to provide interoperability between different *OverArch*-based systems.

References

[1] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, vol. 2735/2003, Berkeley, CA, USA, Feb. 2003, pp. 33–44.

[2] P. Garca, C. Pairot, R. Mondjar, J. Pujol, H. Tejedor, and R. Rallo, "Planetsim: A new overlay network simulation framework," in *Software Engineering and Middleware*, vol. 3437/2005, 2005, pp. 123–136.

[3] C. Jennings, B. B. Lowekamp, E. Rescorla, S. A. Baset, and H. Schulzrinne, "Resource location and discovery (reload)," IETF Internet-Draft, work in progress, draft-ietf-p2psip-base-19, Oct. 2011.

[4] I. Baumgart, B. Heep, and S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework," in *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007*, Anchorage, AK, USA, May 2007, pp. 79–84.

[5] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," in *Peer-to-Peer Systems: First International Workshop (IPTPS 2002). Revised Papers*, vol. 2429/2002, Cambridge, MA, USA, Mar. 2002, pp. 53–65.

[6] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The Bittorrent P2P File-Sharing System: Measurements and Analysis," in *Peer-to-Peer Systems IV*, ser. Lecture Notes in Computer Science, M. Castro and R. van Renesse, Eds. Springer, 2005, vol. 3640, pp. 205–216.

[7] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for Internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, Feb. 2003.

[8] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," in *Middleware 2001 : Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, vol. 2218/2001, Heidelberg, Germany, Nov. 2001, pp. 329–350.

[9] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling Churn in a DHT," in *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, Boston, MA, USA, Jun./Jul. 2004, pp. 127–140.

[10] I. Baumgart and S. Mies, "S/kademlia: A practicable approach towards secure key-based routing," in *Proceedings of the 13th International Conference on Parallel and Distributed Systems (ICPADS '07)*, Hsinchu, Taiwan, Dec. 2007, pp. 1–8.

[11] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," in *First International Conference on Peer-to-Peer Computing (P2P'01)*. Linkpings, Sweden: IEEE Computer Society, Aug. 2001, pp. 99–100.

[12] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making gnutella-like P2P systems scalable," in *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. Karlsruhe, Germany: ACM Press, Aug. 2003, pp. 407–418.

[13] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *SIGCOMM '02: Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Pittsburgh, Pennsylvania, USA: ACM Press, 2002, pp. 205–217.

[14] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. Rowstron, "Scribe: a large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1489–1499, Oct. 2002.

[15] M. Ripeanu, A. Iamnitchi, I. T. Foster, and A. Rogers, "In Search of Simplicity: A Self-Organizing Group Communication Overlay," in *1st IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO '07)*, Boston, MA, USA, Jul. 2007, pp. 371–374.

[16] A.-T. Gai and L. Viennot, "Broose: a Practical Distributed Hashtable based on the De-Bruijn Topology," in *Fourth International Conference on Peer-to-Peer Computing (P2P 2004)*, Zurich, Switzerland, Aug. 2004, pp. 167–174.

[17] M. F. Kaashoek and D. R. Karger, "Koorde: A Simple Degree-Optimal Distributed Hash Table," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, vol. 2735/2003, Berkeley, CA, USA, 2003, pp. 98–107.

[18] I. Baumgart, "P2PNS: A Secure Distributed Name Service for P2PSIP," in *Proceedings of the Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2008)*, Hong Kong, China, Mar. 2008, pp. 480–485.

[19] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "Opendht: a public dht service and its uses," in *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*. Philadelphia, PA, USA: ACM Press, Aug. 2005, pp. 73–84.

[20] M. Wählisch, T. C. Schmidt, and G. Wittenburg, "A Common API for Hybrid Group Communication," in *Proc. of the 34th IEEE Conference on Local Computer Networks (LCN)*, M. Younis and C. T. Chou, Eds. Piscataway, NJ, USA: IEEE Press, October 2009, pp. 265–268.

[21] M. Demmer, K. Fall, T. Koponen, and S. Shenker, "Towards a modern communications api," in *Proceedings of the Sixth Workshop on Hot Topics in Networks (HotNets-VI), Atlanta, Georgia, USA*, Nov 2007.

[22] P. Pietzuch, D. Eyers, S. Kounev, and B. Shand, "Towards a common API for publish/subscribe," in *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, ser. DEBS '07. Toronto, Ontario, Canada: ACM Press, 2007, pp. 152–157.

[23] F. Delmastro, M. Conti, and E. Gregori, "P2P Common API for Structured Overlay Networks: A Cross-Layer Extension," in *Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, ser. WOWMOM '06. Niagara-Falls, Buffalo-NY, USA: IEEE Computer Society, Jun. 2006, pp. 593–597.