2009 IEEE 34th Conference on Local Computer Networks (LCN 2009)
Zürich, Switzerland; 20-23 October 2009

A Common API for Hybrid Group Communication

Matthias Wählisch
FU Berlin, Inst. für Informatik
Takustraße 9
D–14195 Berlin, Germany
Email: waehlisch@ieee.org

Thomas C. Schmidt
HAW Hamburg, Dept. Informatik
Berliner Tor 7
D–20099 Hamburg, Germany
Email: t.schmidt@ieee.org

Georg Wittenburg
FU Berlin, Inst. für Informatik
Takustraße 9
D–14195 Berlin, Germany
Email: wittenbu@inf.fu-berlin.de

***Abstract*—Recent efforts are made to construct a globally accessible group communication service by a simultaneous use of network and application layer multicast. Such hybrid approaches provide native multicast to group members wherever available, but relocate data distribution and duplication from the network to applications or gateways if needed. Such services require an abstract programming interface to allow for a transparent use by applications. The contributions of this paper are twofold. First, we explore the problem space of designing a protocol stack for hybrid group communication that covers structured P2P networks. Second, we propose a transparent API that encapsulates a middleware abstraction layer for implementing hybrid multicast, and allows for overlay-underlay agnostic programming.**

***Index Terms*—Adaptive Group Protocol Stack, Key-based Routing, Inter-domain Hybrid Multicast, Dabek Model**

I. Introduction

Group communication is most efficiently supported on the network layer, but lacks deployment in the global infrastructure. In contrast, overlay multicast remains infrastructure-agnostic, but implements suboptimal forwarding. Hybrid schemes combine both, using native multicast where deployed, and bridge the inter-domain gap by application layer overlays.

Recent approaches to routing overlays have been realized as structured P2P systems and implemented with the help of Distributed Hash Tables (DHTs). Originally, DHTs provide an application layer service that stores and retrieves key-value pairs in a distributed fashion. They are reasonably efficient and scale over a wide range of overlay members in contrast to IP tunneling [1] or unstructured P2P networks. The separation of the store/retrieval function from the peer lookup enables new routing services such as multicast. Structured overlay routing operates on application layer IDs. These keys require a mapping to the underlay in hybrid scenarios.

A modular network stack is needed to transparently interconnect regions of structured overlay and native multicast, as well as an API to control the group management between overlay and underlay. Such a group communication network stack has been introduced in [2]. In this paper, we discuss the problems of flexible, hybrid P2P networks and present a common multicast API. First, we review current work on structured overlay programming (§ II). Second, we introduce our core contribution, the design of a common API for group communication, which serves the requirements of data distribution and maintenance for multicast and broadcast on a middleware abstraction layer (§ III). A final discussion concludes this contribution.

II. Current Concepts of a Common API for Structured Overlay Routing and Multicast

The common way to model routing, services and applications leads to a layered architecture that is composed of flexibly interchangeable modules with common interface definitions between all components. Flexibility emerges from two perspectives: (a) the development process may be simplified as modules can be reused, and (b) deployed modules can be combined from tailored units. In the scenario of structured P2P networks, an application can be composed of overlay implementations best suited for current requirements on performance and service needs. This may not only reduce complexity, but also diminish maintenance overhead from unwanted services. In the following, we discuss the common model to decompose DHT systems, and review related work.

A. The Dabek Model

The first ideas towards a layered architecture with a common API for structured overlays have been presented in [3]. The concept is known as the Dabek model and has been implemented in numerous simulators [4] and DHT stacks [5]. The basis of the Dabek model is a unified overlay routing interface.

The decentralized routing mechanism of P2P protocols serves as a virtualized network layer. Dabek *et al.* observed that many peer-to-peer services and applications are built upon a key-based routing (KBR) module, which can locate peers for multiple purposes. Based there on, the authors suggest a compound P2P layer consisting of three tiers: Tier 0 represents the fundament of overlay communication, the KBR layer. Tier 1 implements higher level abstractions, complemented by tier 2 for end user applications and further, higher level services.

The Dabek model focuses on tier 0. The meaning of tier 1 and 2 has not been fully elaborated. Both tiers may accommodate services with direct access to the KBR layer. Reusable abstractions for specific purposes are dedicated to reside on tier 1. The DHT abstraction is such an example for which provides a store-and-retrieval key management service on the overlay for applications like file sharing. On top of the DHT abstraction, further services may be available, such as an additional caching at 'end-user' applications or specific store and retrieval functions.

1) The Common Key-based Routing API: The key-based routing layer provides the option to forward data along overlay hops. With respect to the concepts of layered architectures, an

implementation may delegate messages to the KBR component that performs the routing. Hence, all application data will be encapsulated in overlay messages and delivered by the KBR. However, this may be inefficient, because it can cause double encapsulation, at the service and the KBR layer. Additionally, a tier 1 service may follow a different routing strategy than the overlay paths. Thus, a common API should remain open with respect to adaptable message forwarding, as well as KBR state access.

The KBR API by Dabek *et al.* consists of two functional groups. The first part provides function calls for sending and receiving data messages from above tier 0, and also for controlling the overlay routing path for messages initiated by the service or application. These are subsumed as message routing. The second part grants access to local KBR routing states.

For a detailed description of the routing and state access functions, we refer to [3]. Applications can invoke these functions from higher tiers to inquire about the *local* peer routing states. Nevertheless, local lookups will fail, whenever the requested information is not locally available. To facilitate a global peer access, the message routing functions can be used. The routing as well as the state access functions require a common `key` parameter that corresponds to the overlay address. Consequently, all applications using one of the primitives must be aware of the hash function in use.

2) Limitations of the KBR-API: The generic approach of the KBR-API does not provide information about the actual overlay routing protocol or implementation-specific parameters, such as the key properties in KBR or the dimension in the DHT CAN [6] (Content Addressable Network). Such meta information is of interest for services that implement cross layering or operate by adapting to the underlying KBR.

An example is given by specilized routing services that construct individual forwarding tables from local state information. Using the common API, these services can retrieve generic destination values, but are not enabled to reconstruct the underlying routing structure.

To make the protocol parameters visible to upper tiers, the API needs extensions. One possible concept could include dynamic maps to present the KBR specific configurations in the form of key value pairs. The corresponding schemes can be implemented by information bases similar to Management Information Bases (MIBs). The implementation of this approach requires only a getter call. The actual parameter set is then defined by the protocol instance.

In using such a rich, compound P2P layer concept, existing key-based routing implementations can be enhanced by new services without changing the KBR component. For this purpose, a new service is only required to implement the common KBR calls. However, more complex services such as overlay multicast need to provide their own common API towards applications.

B. Group Service Models

The Dabek model proposes a multicast abstraction, called CAST [3]. The idea is to provide overlay services in a generic multicast module. CAST consists of a set of interfaces for group management and data distribution (`join(groupId)`, `leave(groupId)`, `multicast/anycast(msg,groupId)`), as well as a basic multicast routing on top of the KBR layer. Routing within CAST is built upon a dedicated tree management and forwarding scheme, which is similar to Scribe [7] (a PIM-SM like overlay approach). Calling the `join` function initiates a subscription message routed towards the hash of the group id, employing the KBR `route`. At all intermediate peers, the upcall `forward` is invoked to establish corresponding multicast states in CAST. On top of CAST, the authors have foreseen further multicast implementations, e.g., Scribe or Bayeux [8].

This approach of a universal routing protocol as part of the middleware layer may conflict with the forwarding strategy pursued by services above CAST. A simple example can be identified in the difference between Scribe and Bayeux. While Scribe creates a rendezvous point-based shared tree according to reverse path forwarding, Bayeux sets up source-specific states along the path from the source to the new receiver. Thus for Bayeux, CAST would establish multicast states in the opposite direction. Indeed, the idea of providing the P2P layer with a generic multicast routing logic is valuable for applications agnostic to routing services, but fails in general for overlay multicast (OLM) modules. Group specific routing forms the core component of any multicast solution and may differ by approach and domain-specific demands. In this sense, a service abstraction should only provide an interface definition, but not a routing logic.

Another application layer multicast (ALM) middleware architecture including a wrapper API is currently presented in [9]. The authors assume that an ALM protocol consists of the following parts: group management, topology management and traffic management, which can be adopted by different ALM protocols. For the latter, they propose an API for interoperability and transparency. The API calls support the selection of different transport protocols, as well as a native network and an overlay transmission mode.

The motivation of the suggested API is to provide a unique interface for supporting structured and unstructured ALM protocols. As mentioned by the authors, the requirements slightly differ, which is reflected in the API design. Unstructured group management introduces functions unknown in the context of structured overlay multicast, an external group management for example. Many unstructured multicast schemes rely on central management and provide a global view on the group structure. In general, the simultaneous support of centralized and decentralized approaches poses a severe challenge to a common middleware. Such a cooperative deployment scenario is less likely than KBR protocols jointly operating with network layer multicast, as both are fully distributed and explicitly neglect a server infrastructure.

The authors in [10] present a middleware for unstructured application layer multicast only. They decompose the ALM component into several functional units, e.g., a metric esti-

mator or a logic net to maintain and optimize the overlay network. The proposed API does not account for a transparent overlay and underlay group communication and consists only of simple receive/send calls.

III. A Middleware for Structured P2P Group Communication

A. Objectives

The main task of an overlay multicast (OLM) component is the distribution of data according to the host group model. Destined for a group address, messages are replicated along an existing (virtual) routing infrastructure similar to native networks. On the unicast side, routing in structured overlays can be provided by the KBR layer and the access may be decoupled from a specific implementation using the common primitives. Thus, structured overlay group communication requires at least the KBR layer.

The KBR layer connects peers to a unicast network. Multicast domains will be established, when overlay nodes form a group as maintained within the OLM middleware. A special case is broadcast, which inherently floods all nodes of the network. This may happen without the establishment of selected routing paths. Hence, multicast and broadcast follow different distribution schemes, and require distinction at the API level. A modular OLM stack should account for these conceptual different challenges and solutions.

Besides flexibility, an OLM design should also be guided by a compatibility principle with respect to native network services. On the one hand, the API calls, e.g., joining and leaving groups, should be compliant to well-known functions, such that application developers are not distracted. On the other hand, the OLM middleware should provide an interface that allows for transmission of data to both overlay and native multicast networks.

Overlay group communication differs from native IP networks with respect to addressing. An IP stack is fed with the correct network address type to perform routing, whereas the key-based routing layer maps an arbitrary identifier into the deployed key space, which commonly is not invertible and does not allow to recover original IDs. Applications need to regain those identifiers explicitly, and it may be an advantage for the overlay routing to be aware of the application addresses, as well. Based on application addresses, the overlay scheme may for instance aggregate or scope groups. Thus, the design of an OLM middleware should preserve common group functions, but also include support for specific aspects of the layers involved, and remain open for additional functionalities arising from structured overlay networks.

B. Current Challenges

The APIs presented so far either focus on a common interface for tree construction, or a direct adoption of native group management calls. However, the concepts do not account for two further important aspects:

(a) Which types of addresses may join the application?
(b) How does the OLM API provide dedicated broadcast?

Commonly, a structured overlay network does not restrict the address space to any specific type. Each address will be handled as a string and hashed to the same identifier space without further syntactical or semantical processing. Nevertheless, the overlay may require special addresses for group communication to predefine a subset of group members, such as a broadcast address. Applications operate in different contexts and denote communication parties with respect to a domain-specific namespace. Special addresses should be available in all namespaces to allow for its continuous use.

Native IP and many structured OLM schemes offer a native broadcast service, and supplementing the API with an additional broadcast interface appears as a simple solution for service access. However, broadcast and multicast operate on the same overlay, which may result in key collisions. Consequently, the OLM middleware should foresee a specific, but well-known broadcast address, which can also be used to identify broadcast and multicast data on the same channel. There are two natural options to map broadcast addresses uniquely onto the same overlay key. The API may define a broadcast address that belongs to a specific context, but is obligatory for all applications. Application programmers then have to use this specific address and may need to account for context switches. Alternatively, broadcast addresses should be embedded in every namespace. These multiple, dedicated addresses can then be mapped by the middleware to a common identifier, which does not conflict with multicast addresses. For this reason, an OLM should be aware of namespaces, and each namespace should include a unique broadcast address. This can be implemented, e.g., by using the natural $255.255.255.255$ for IPv4 or the link-local all-nodes address for the IPv6 namespace. Application layer addresses such as SIP URLs can reserve the asterisk for broadcast identification.

Address awareness may enhance overlay group communication over native multicast. A typical example may be news channels, e.g., sam@irtf.org and mobopts@irtf.org, which fall into a combined group *@irtf.org. Based on a corresponding aggregation, a user subscribes only once, instead of joining each channel. Such group aggregation can be applied, if an arbitrary namespace includes a broadcast address and the OLM component is semantically aware of the namespace definition. In the described example, the OLM middleware would identify the user and host parts of the compound namespace and initiate a partial broadcast to all members of irtf.org.[1]

C. An API Proposal

The architecture of a hybrid group communication stack and the interplay of overlay and IP multicast routing is described in [2]. In the following, we introduce the corresponding API.

Overlay routing is based on hashed keys. As applications are unaware of any overlay specifics, the mapping of the

[1]This group aggregation does not follow the common multicast paradigm, but can for instance be naturally implemented in CAN which we only sketch: Each part of the namespace is separately hashed and corresponds to a CAN dimension. Equal addresses result in equal CAN coordinates. Flooding the selected namespace then corresponds to data dissemination in the selected dimension.

destination address to the key space should be performed on the ALM layer as suggested by Dabek *et al*. Nevertheless, it is important to preserve the original group identifier, as an application may receive multiple streams of different multicast addresses via the same communication channel. As an overlay is used in a specific context, the identifiers selected by the applications are likely to belong to the same namespace. Thus, we suggest to pre-initialize the communication channel between application and group stack with the corresponding context to simplify the API calls.

The destination group address configured by the application is a common application layer or network address and is denoted by `address`. In contrast, overlay IDs are identified as `key`. The application can choose, if it sends the data to the underlay or overlay. We denote the corresponding data type `mode`, which also allows leaving the decision to the group communication stack, if the mode is unspecified. In the following, we will describe the API calls used between the group communication stack and the application.

At first, we explain briefly calls to function for sending and receiving multicast data, thus, reflecting typical source and receiver instances.

`init(namespace→n)` This call is implemented by the multicast middleware to set up the communication channel between application and stack. It preinitializes the namespace, which then can be used for all further calls.

`void join(address→a, mode→m)` The join call is implemented by the ALM stack. It initiates a group subscription. Depending on the mode, this may result in an IGMP/MLD join, if the address is a valid IP multicast address. The address of joins towards the overlay will be pre-processed to implement, e.g., group aggregation and broadcast. The middleware creates a corresponding overlay key.

`void leave(address→a, mode→m)` This downcall is implemented by the middleware and results in an unsubscription for the given address.

`void send(address→a, mode→m, message→msg)` This function is invoked at the middleware to send group data. If the overlay parameter has been configured, the middleware decides to forward the data supplied with the corresponding key to the broadcast or multicast module based on the destination address.

`void receive(address→a, message→msg)` This upcall is implemented by the application and delivers overlay and underlay messages received at the node. The address represents the destination used by the source application instance.

To request multicast states, we define the following group service API:

`nodehandle [] groupSet(mode→m)` This operation returns all registered multicast groups. The information can be provided by group management or routing protocols. The return values distinguish between sender and listener states.

`nodehandle [] neighborSet(mode→m)` This function can be invoked at the middleware to get the set of multicast routing neighbors.

`bool designatedHost(address→a)` This function is implemented by the middleware and returns true, if the host has the role of a designated forwarder or querier. Such an information is provided by almost all multicast protocols to handle packet duplication, if multiple multicast instances serve on the same subnet.

`address updateListener(mode→m)` This upcall is invoked to inform a group service about a change of listener states for a group. This is the result of receiver new subscriptions or leaves. The group service may call `groupSet` to get updated information.

`address updateSender(mode→m)` This upcall is implemented by the middleware to inform the application about source state changes. Analog to the `updateListener` case, the group service may call thereupon `groupSet`.

IV. Conclusions

Hybrid multicast combining overlay and underlay routing is commonly seen as a promising path to a widening deployment of group communication services. Its transparent use by arbitrary applications will be fostered by a common programming interface present at applications or infrastructural gateways. In this paper, we have presented and discussed such a universal group API, which along the way opens opportunities for enriching group communication functions beyond the basic host group model. The API can be applied immediately to implement inter-domain multicast gateways [2], which may serve as the building blocks in a future hybrid shared multicast architecture. In future work, we plan for real deployment experiments with a special focus on inter-provider interaction.

References

[1] D. Thaler, M. Talwar, A. Aggarwal, L. Vicisano, and T. Pusateri, "Automatic IP Multicast Without Explicit Tunnels (AMT)," IETF, Internet Draft – work in progress 09, Jun. 2008.

[2] M. Wählisch, T. C. Schmidt, and G. Wittenburg, "A Generalized Group Communication Network Stack and its Application to Hybrid Multicast," in *Proceedings of the 28th IEEE INFOCOM. Student Workshop*. IEEE Press, April 2009.

[3] F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," in *Proc. of IPTPS 2003*, ser. LNCS, M. F. Kaashoek and I. Stoica, Eds., vol. 2735. Berlin Heidelberg: Springer–Verlag, 2003, pp. 33–44.

[4] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers, "The State of Peer-to-Peer Simulators and Simulations," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 2, pp. 95–98, 2007.

[5] P. Druschel *et al.*, "FreePastry," http://freepastry.rice.edu/FreePastry/.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A Scalable Content-Addressable Network," in *Proc. of SIGCOMM 2001*. New York, NY, USA: ACM, 2001, pp. 161–172.

[7] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "SCRIBE: A large-scale and decentralized application-level multicast infrastructure," *IEEE JSAC*, vol. 20, no. 8, pp. 100–110, 2002.

[8] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz, "Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination," in *Proc. of NOSSDAV 2001*, New York, NY, USA: ACM, 2001, pp. 11–20.

[9] B. P. Lim and K. Ettikan, "ALM API for Topology Management and Network Layer Transparent Multimedia Transport," IETF, Internet Draft – work in progress 00, Jan. 2008.

[10] N. Mimura, K. Nakauchi, H. Morikawa, and T. Aoyama, "RelayCast: A Middleware for Application-level Multicast Services," in *Proc. of CCGRID 2003*, Washington: IEEE Comp. Soc., 2003, pp. 434–441.