GossipKit: A Framework of Gossip Protocol Family

Shen Lin, François Taïani, Gordon S. Blair
Computing Department
Lancaster University
Lancaster LA1 4YR, UK
(s.lin6, f.taiani, gordon)@comp.lancs.ac.uk

***Abstract*—A large number of gossip protocols have been developed in the last few years to address a wide range of functionalities. So far, however, very few software frameworks have been proposed to ease the development and deployment of these gossip protocols. To address this issue, this paper presents GossipKit, an event-driven framework that provides a generic and extensible architecture for the development of (re)configurable gossip-oriented middleware. GossipKit is based on a generic interaction model for gossip protocols and relies on a fine-grained event mechanism to facilitate configuration and reconfiguration and promote code reuse.**

1. Introduction and Problem Statement

Gossip-based algorithms have recently become extremely popular. The underlying concept of these algorithms is that individual nodes repeatedly exchange data with some randomly selected neighbours, causing information to eventually spread through the system in a "rumour-like" fashion. Gossip-based protocols offer several key advantages over more traditional systems: 1) they provide a scalable approach to communication in very large systems; 2) thanks to the randomised and periodic exchange of information, they offer self-healing capacities and robustness to failures; and 3) they are simple to implement. Because of these benefits, gossip-based protocols have been applied to a wide range of contexts such as peer sampling [9], [17], ad-hoc routing [14], reliable multicast [1], [2], database replication [10], failure detection [11], and data aggregation [12].

Unfortunately, past research has mainly focused on the development and evaluation of new gossip protocols. In particular very few attempts have been made at developing (re)configurable middleware architectures to support gossip-based systems. T-Man [5] and the recent work at Bologna [6] are two of the early gossip-dedicated frameworks that have been proposed in this area. They both rely on a common periodic gossip pattern to support a variety of gossip protocols. Although these frameworks can help develop gossip-based systems to a significant extent, we contend that they only partially address the issues faced by the developers of gossip-based applications. First, the common periodic gossip pattern they rely on only captures the features of proactive gossip protocols. As such, it does not support reactive gossip algorithms. Second, these frameworks tend to be monolithic and as such do not provide a flexible architecture that is easily extensible. Third, these frameworks do not support runtime reconfiguration.

This paper introduces GossipKit, a fine-grained event-driven framework we have developed to ease the development of (re)configurable gossip-based systems that operate in heterogeneous networks such as IP-based networks and mobile ad-hoc networks. The goal of GossipKit is to provide a middleware toolkit that helps programmers and system designers develop, deploy, and maintain distributed gossip-oriented applications. GossipKit has a component-based architecture that promotes code reuse and facilitates the development of new protocols. By enforcing the same structure across multiple and possibly co-existing protocols, GossipKit simplifies the deployment and configuration of multiple protocol instances. Finally, at runtime, GossipKit allows multiple protocol instances to be dynamically loaded, operate concurrently, and collaborate with each other in order to achieve more sophisticated operations.

The contributions of this paper are threefold. First, we identify a generic and modular interaction pattern that most gossip protocols follow. Second, we propose an event-driven architecture based on this pattern that can be easily extended to cover a wider range of gossip protocols. Third, we briefly evaluate how our event-driven architecture provides a fine-grained mechanism to compose gossip protocols within the GossipKit framework.

The remainder of the paper is organised as follows. Section 2 discusses related work. Section 3 presents a study of existing gossip protocols and explains how this study informed the key design choices of GossipKit. Section 4 gives an overview of GossipKit's architecture. Section 5 describes the current implementation of the GossipKit framework, while an early evaluation is provided in Section 6. Finally, Section 7 concludes the paper and points out future work.

2. Related Work

Two categories of communication frameworks have been proposed to support gossip protocols: Gossip Frameworks, which explicitly and directly support gossip-based systems, and Event-driven communication systems, which tend to be more generic and more flexible. In this section we analyse the strengths and weaknesses of both of them from the viewpoint of gossip protocol development.

Gossip frameworks are specifically designed to support gossip protocols. Typical examples of such framework are T-Man [5] and the recent work on this topic at Bologna [6]. These two frameworks assume that most gossip protocols adopt a common proactive gossip pattern. In this gossip pattern, a peer P maintains two threads. One is an active thread, which periodically pushes the local state S_P to a randomly selected peer Q or pulls for Q's local state S_Q. The other is passive, which listens to push or pull messages from other peers. If the received message is pull, P replies with S_P; if the received message is push, P updates S_P with the state in the message.

To develop a new gossip protocol within this common proactive gossip pattern, one only needs to define a state S, a method of peer selection, an interaction style (i.e. pull, push or pull-push), and a state update method. Many proactive gossip protocols such as peer sampling service, data aggregation, and topologic maintenance have been implemented in such Gossip frameworks.

However, the monolithic design of these Gossip frameworks makes them inadapted to protocols that use a reactive gossip pattern (e.g. SCAMP [9]) or those implementing sophisticated optimisations such as feedback based dissemination decision [13] and premature gossip death prevention [14]. Furthermore, these Gossip frameworks neither support reconfiguration nor concurrent operation of multiple gossip protocols at runtime.

Event-driven communication systems aim to provide a flexible composition model based on event-driven execution. They are

developed to support general-purpose communication and but not specifically for gossip protocols. Examples of such communication systems are Ensemble [3], Cactus [4] and their predecessors Isis [7] and Coyote [8]. In these environments, a configurable service (e.g. a Configurable Transport Protocol) is viewed as a composition of several functional properties (e.g. reliability, flow control, and ordering). Each functional property is then implemented as a micro-protocol that consists of a collection of event handlers. Multiple event handlers may be bound to a particular event and when this event occurs, all bounded event handlers are executed.

Event-driven communication systems offer a number of benefits for developing gossip protocols. First, individual micro-protocols can be reused to construct families of related gossip protocols (implemented as services) for different applications instead of implementing a new service from scratch for each protocol. Second, reconfigurability can be achieved by dynamically loading micro-protocols and rebinding event handlers to appropriate events. Finally, the use of event handlers present a fine-grained decomposition of protocols.

However, event-driven frameworks are known to be notoriously difficult to program and configure as argued in [16]. In large part, this is because these frameworks do not by themselves include any domain-specific features (e.g. interaction patterns and common structure) for individual protocol types.

In order to address the major shortcomings discussed in this section, GossipKit adopts *a hybrid approach that combines domain-specific abstraction and the strengths of event-driven architecture*. The remaining sections of this paper present its design and prototype implementation.

3. GossipKit's Key Design Choices

To design GossipKit, we first investigated a number of existing gossip-based protocols and identified similarities and differences amongst them. In this section, we report on the results of this study and present the key design choices we made for GossipKit based on these results. More precisely we look at three aspects of gossip protocols: Section 3.1 explains the reason of using domain-specific interfaces for different types of gossip protocol to interface with external applications. Section 3.2 presents the common interaction pattern of gossip protocols that we have observed, and finally Section 3.3 argues the benefits of adopting an event-driven architecture for our gossip protocol framework.

3.1 Application-dependent Interfaces

As mentioned previously, gossip-based solutions have been proposed for a wide range of distributed applications. Different types of gossip protocol interact with the external world distinctively. For instance, a gossip-based routing protocol must provide an interface for external application systems to trigger the route request that will be gossiped, whilst a gossip protocol for peer sampling service needs to provide access to the collected peer samples. From our experience and analysis, it is unlikely to identify a common generic interface that can separate gossip protocols from the applications that utilise them. Instead we proposed to identify a *set* of generic but domain-specific interfaces that can each support a family of gossip protocols in a particular application domain. In order to do so, we have classified gossip protocols into categories in accordance with their functionality. This has enabled us to identify a common interface for gossip protocols within each category that can be used to interact with their external applications. Through domain-specific common interfaces, external applications can access various types of gossip protocols that operate in a single framework. Section 4.1 will describe the mapping between these domain-specific interfaces and control logic in detail.

3.2 Common Interaction Pattern

Although different types of gossip protocols provide divergent interfaces to external applications, we have found that, internally, they all follow the same interaction pattern. This common interaction pattern can be captured using a modular approach and combines the proactive gossip pattern that has been identified in [5] and [6], with the reactive gossip patterns observed on gossip protocols such as [9] and [14]. This common interaction model is shown in Fig. 1. In this figure, the modules involved in the interaction are presented as boxes, and interactions between modules as arrowed lines. The direction of the arrows indicates which module initiates the interaction, and the labels show in which sequence these interactions take place.

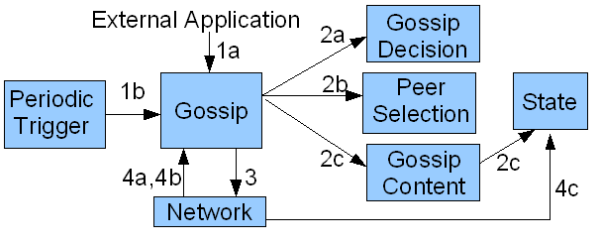

Fig. 1. Common Interaction Model

Initially, a gossip dissemination can either be raised periodically (e.g. a periodic pull or push of gossip message), or upon a receipt of an external request (e.g. an ad-hoc routing protocol requesting a reactive gossip protocol such as [14] to gossip a route request). These two interactions are represented as 1a and 1b in Fig. 1, respectively.

The second phase prepares the gossip action. Some gossip protocols may use various policies to decide whether to gossip at the current situation (2a). For instance, a reactive gossip protocol may decide not to gossip the same message twice or forward the message with a given probability. If a decision is made to forward the gossip message, the protocol instance must then select the peers it wishes to gossip with (2b). In addition, many gossip protocols will need to decide which content is to be gossiped (2c). In particular, a proactive gossip protocol typically requires to retrieve the gossip content from its local state if it needs to send periodically its state (push-style gossip) or reply to a request of its state (pull-style gossip). An example of gossip content could be the temperature sensed by each peer.

The third phase is gossip dissemination. It utilises the underlying network to send gossip messages to the selected peers (3).

Finally, on receipt of a gossip message from the network, a gossip protocol may react in three different ways, depending on the type of the received message: 1) it might forward the message to peers that it knows (4a) and this may involve the interactions in phase 2 (2a, 2b and 2c); 2) it might respond with its own state (4b) and similarly this can involve the interactions in phase 2; and 3) it might extract the state contained in the message and merges with its own state (4c).

Note that this overall interaction model can be invoked recursively — each module presented in Fig. 1 can itself be implemented as a gossip protocol that follows the interaction model. For instance, the Peer Selection module can be a gossip-based peer sampling service protocol.

In practice, various gossip protocols may be composed from completely different implementations of modules in Fig. 1, and these

coarse-grained modules can hardly be reused. In order to enable optimal reuse, the framework allows each module to be composed from a variety of finer-grained micro-modules.

More precisely, we have noticed that five modules (Gossip, Peer Selection, Gossip Decision, Gossip Content, and State) in Fig. 1 can often be decomposed into finer-grained and reusable *micro-modules*. Each individual micro-module implements a distinct algorithm, and different combinations of these micro-modules can form modules with more sophisticated behaviours. Consider the example presented in Fig. 2. This example shows three gossip-decision policies used in a gossip-based ad-hoc routing protocols ($Gossip1(p)$, $Gossip2(p,k)$, and $Gossip3(p,k,p_1,n)$) [14]. Instead of being implemented as independent coarse-grained decision modules, these three decision strategies can reuse the same three fine-grained micro-modules.

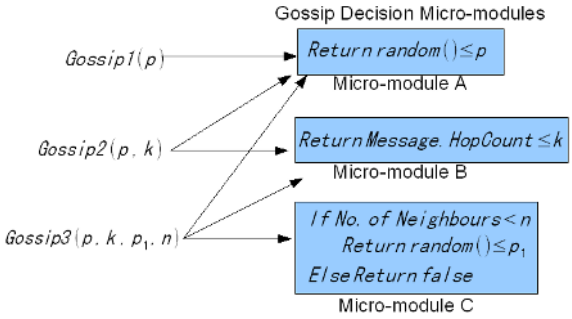

Fig. 2. Various Gossip Decision modules realised by different composition of micro-modules

More precisely, $Gossip1$, $Gossip2$, and $Gossip3$ differ by how they decide whether to forward the received routing request message (i.e. they require different versions of the Gossip Decision module): $Gossip1$ forwards the message with probability p; $Gossip2$ is the same as $Gossip1$ except that it forwards the message with probability 1 in the fist k hops; and $Gossip3$ is the same as $Gossip2$ except that it forwards message with probability $p_1 > p$ if it has less than n neighbouring peers.

These three different gossip decision strategies can be implemented by different combination of the three fine-grained Gossip Decision micro-modules shown on Fig. 2. $Gossip1$ can directly use micro-module A as its Gossip Decision module; $Gossip2$'s Gossip Decision module can be viewed as a composition of micro-module A and B by evaluating the return values of these two micro-module using boolean operation OR to obtain the decision for forwarding the message; and $Gossip3$'s Gossip Decision module can be composed from micro-module A, B, and C in the same way as $Gossip2$ does.

3.3 Event-driven Architecture

The common interaction pattern of gossip protocols we have just presented serves as the basis for our architecture design. Based on the study of gossip protocols, it is clear that a generic system architecture should satisfy the following two criteria.

First, our architecture should allow micro-modules to be easily configured and implement the various modules found in our common interaction pattern. This requirement can be fulfilled using event-driven frameworks such as Ensemble and Cactus. In these frameworks, micro-modules (e.g. Gossip Decision micro-modules shown in Fig. 2) can be viewed as event handlers that are bound to certain events, and the arbitrary composition of micro-modules can be simplified to uniform event-bindings. For instance, to compose a Gossip Decision module, Gossip Decision micro-modules can be bound to events raised by Gossip modules (Gossip Decision module is invoked by Gossip module as shown in Fig. 1). The Gossip module then evaluates the return values of the invoked Gossip Decision micro-modules using boolean operation OR, so as to obtain the decision for forwarding the message. Furthermore, one can simply change the event-bindings to obtain a different composition of micro-modules.

Second, the architecture should be easily extensible to support new gossip protocols on the basis of the common interaction pattern shown in Fig. 1. This is because our interaction pattern is based on the study of typical and representative gossip protocols. It does not cover however all existing gossip algorithms. New gossip protocols may require extra modules and interactions beyond the common interaction pattern. Therefore, it is important that the system architecture allows new modules and interactions to be added onto the pattern. This issue can be addressed by using event-driven systems. In an event-driven system, interactions between event handlers can be achieved through passing events and hence, minimises the explicit references between modules as argued in [8]. As a consequence, our framework can be easily extended by plugging in new micro-modules (i.e. event handlers) and reconfiguring the event binding to support new interaction patterns.

From the above analysis, we have therefore chosen an event-driven architecture for our framework in order to easily configure the composition of micro-modules and to improve extensibility of the common interaction model in Fig. 1. The details of the resulting architecture are presented in Section 4.

4. GossipKit's Architectural Overview

Our architecture consists of five components as shown in Fig. 3. In the figure, an interaction between two components is represented as a pair of connected interface and receptacle. The API components implement the domain-specific interfaces described in Section 3.1. The remaining components realise the common interaction pattern described in Section 3.2. The remainder of this section discusses these components and their interactions in detail.

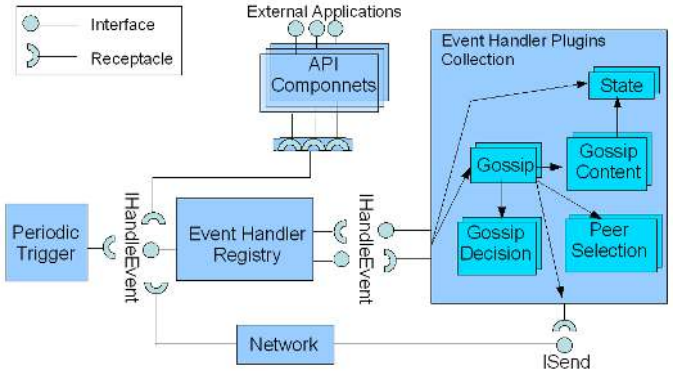

Fig. 3. GossipKit Architecture

4.1 API Components

API components aim to uncouple the gossip protocols implemented by the framework from external applications. Each type of API component provides a generic interface for external applications to access a particular category of gossip protocols. When an interface of an API component is triggered by the connected external application, it raises an event to the event handler registry. Fig. 4 provides an example of how API component interacts with external applications.

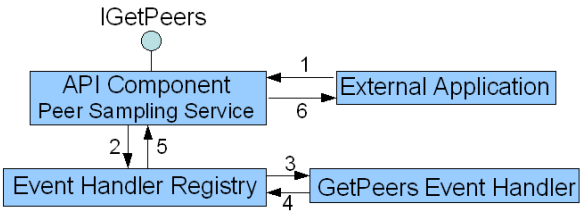

Fig. 4. Interaction of API Component with External Application

This figure shows the API component for peer sampling service protocols. This API component provides an IGetPeers interface for external application or other API components to retrieve peer information collected by the local peer. When IGetPeers is invoked (operation 1 in Fig. 4), the API component generates a GetPeers event to the event handler registry (operation 2). On receiving this event, the registry executes the proper event handler to handle the GetPeers event (operation 3, see section 4.3 below). The event handler then retrieves the peer sampling information stored locally, and returns the information to the API component as the event handling result (operation 4 and 5). Finally, the API component provides the peer sampling information to the external application as the return value of the IGetPeers interface (operation 6).

4.2 Periodic Trigger Component

The periodic trigger component is optional in the framework. It is only loaded when the framework is used to support proactive gossip protocols. This component periodically dispatches events to trigger specific event handlers that perform different styles of gossiping, such as pull, push or pull-push. The event-dispatching period (the gossip frequency) is predetermined at deployment phase, and can be reconfigured at runtime.

4.3 Event Handler Registry

The event handler registry serves as a broker between event handlers and event producers (components that raise events). The event handler registry maintains a table that records event handler IDs with their associated events (i.e. events that an event handler can handle). When an event handler's IHandleEvent interface is connected to the registry, the registry's table records the events bound to the event handler. The event handler registry also provides an IHandleEvent interface to event producers to trigger the events. On the invocation of an event, the event registry finds and executes the registered event handlers that are bound to this particular event type.

It is worth pointing out that the IHandleEvent interface can also be used by the event handlers themselves. This allows events raised internally within an event handler to be handled by others, thus providing a consistent event-based environment and facilitating interoperability between different gossip protocols.

4.4 Event Handler Plugins

As mentioned in Section 3.3, we considered modules that can be further decomposed to finer-grained micro-modules (i.e. Gossip, Peers Selection, Gossip Decision, Gossip Content, and State in Fig. 1) to be developed as a collection of event handlers. This is reflected by the event handler plugins in Fig. 3. In the figure, multiple micro-modules belonging to each particular module are designed as event handler plugins that are contained in the event handler plugin collection. Micro-modules for the Gossip module and the State module can be invoked by the event handler registry to handle events generated by the periodic trigger component, the API components, and the network component (see below). Micro-modules for the Gossip module can also send messages using the interface provided by the network component. Furthermore, each micro-module can invoke the IHandleEvent interface provided by the event handle registry to interoperate with other micro-modules.

4.5 Network Component

This component provides network level communication to other components, and as such is responsible both for sending messages generated by the Gossip module and for delivering message events received from the network to the event handler registry. Through this component, gossip protocols within the GossipKit framework can operate on transport layers such as UDP, TCP, or ad-hoc routing. The network component can also operate on virtual transport layers in order to utilise the features provided by various component-based virtual overlays such as GridKit [19].

5. Implementation

GossipKit's prototype implementation is based on the Java version of OpenCom [15], a lightweight, efficient and reflective component model. Java's portability enables GossipKit to operate on various platforms, from desktop computers through to PDA. We implemented the micro-modules and event handler plugins shown in Fig. 3 as individual OpenCom components, while we realised events with a normal Java class. This class contains: (i) a header that identifies the type of the event, (ii) a body containing data to be handled by the corresponding event handlers, (iii) a source ID identifying the peer that generated the event, and (iv) a target ID that defines the target peer that should receive the event.

It is worth emphasising the implementation of the periodic trigger component, which can be viewed as a task scheduler that can be utilised by multiple protocols to perform periodic gossiping with different frequencies. Its implementation only requires a single Java thread rather than spawning one thread for each proactive gossip protocol. If multiple proactive gossip protocols operate concurrently at runtime, the resource utilisation of the system can be significantly improved by minimising the use of resource-consuming multi-threading. This effectively reduces memory usage if GossipKit operates on mobile devices that are resource constraint.

6. Early Evaluation

We evaluated our GossipKit framework on two categories of gossip protocols: We implemented three peer-sampling services (SCAMP [9], PSS [17], and the topologic construction protocol described in T-Man [5]), and two reliable multicast protocols (Bimodal Multicast [2], and Lpbcast [1]). In the following, we focus our evaluation on the reusability of the GossipKit framework (Section 6.1). We then briefly discuss the configurability and reconfigurability of our framework in Section 6.2.

6.1 Reusability

We evaluated the reusability of GossipKit using a quantitative measuring approach suggested in [18]. This approach measures the size of the Java classes that make up different configurations of components. In Fig. 5, the first three configurations indicate the cost of each individual protocol in the framework (a tick means the protocol is selected in the configuration). The size of configuring multiple protocols is measured in Configurations 4-6. These measurements are compared against the side-by-side measurement of individual protocols. It can be seen that compiled Java code size is reduced by about 33% in Configuration 4 and 5, and 48% in Configuration 6. These results show that the GossipKit framework does not only promote code reuse for developing gossip protocols that belong to the same

category (SCAMP and PPS in Configuration 4 belong to the peer sampling category), but also for those belong to different categories (PPS and Bimodal Multicast in Configuration 5). Furthermore, the evaluation results indicate the reusable quantity increases as more gossip protocols are deployed in GossipKit (Configuration 6).

Protocols / Config.	SCAMP	PPS	Bimodal Multicast	Framework Size (KB)	Side by Side Size (KB)
Config 1	✓			20.39	
Config 2		✓		26.57	
Config 3			✓	37.31	
Config 4	✓	✓		31.19	46.96
Config 5		✓	✓	38.92	63.88
Config 6	✓	✓	✓	43.55	84.27

Fig. 5. Reusability Measurement.

6.2 Configurability and Reconfigurability

GossipKit offers a common component architecture to simplify the configuration of gossip-oriented middleware. It does so by providing module types and connection bindings between modules that remain the same regardless of the implemented protocols. However, the use of fine-grained micro-modules in GossipKit's event-driven architecture can make configuration a time-consuming process. Although an event-driven architecture simplifies the configuration of micro-modules into modules as discussed in Section 3.3, the manual configuration of event bindings for a large number of micro-modules still remains a time-consuming task, in particular when a user needs to deploy a number of gossip protocols to operate concurrently within GossipKit. From our experiences on the development of five gossip protocols, we have noticed that GossipKit eases the configuration process for these gossip protocols to a certain level. However, further study is required to evaluate whether GossipKit can support easy configuration of a broader range of gossip protocols.

GossipKit supports fine-grained reconfiguration to adapt to environmental changes — different protocol behaviours can be achieved by replacing a simple single component. For instance, a proactive gossip protocol that provides peer sampling service can be modified to support number averaging by replacing the stateful event handler, and the network component that supports communication for multiple gossip protocols can be replaced by another routing scheme. This form of component replacement relies on the mechanisms directly provided by OpenCOM. A detailed discussion of these mechanisms is however out of the scope of this paper.

7. Conclusion and Future Work

This paper has presented GossipKit, an event-based gossip protocol framework. This framework aims to facilitate the development of configurable and reconfigurable middleware that supports multiple gossip protocols potentially operating in parallel under different types of networks. We have presented an early prototype implemented using a reflective component model (OpenCom), and we have discussed some of the benefits we have observed when implementing several gossip protocols with our framework. Our early evaluation indicates that GossipKit promotes code reuse, simplifies configuration for deploying gossip protocol middleware, reduces the overhead for runtime reconfiguration, and minimises the resource usage at runtime to a certain level.

In the future, we plan to explore a broader range of gossip protocols in order to identify more domain-specific features and to improve the genericity of the common interaction model. We are also currently building a configuration tool to allow users to describe a selection and composition of micro-modules, and to automatically configure event bindings of event handlers in order to address the issue discussed in Section 6.2. Furthermore, we plan to utilise the self-organising features of gossip protocols to improve GossipKit towards a self-adaptive framework so that it can automatically reconfigure itself and adapt to changes in its environment.

References

[1] P. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, and P. Kouznetsov, *Lightweight Probabilistic Broadcast*. In IEEE International Conference on Dependable Systems and Networks(DSN2001), July 2001.

[2] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu and Y. Minsky, *Bimodal multicast*. TR99-1745, May 11, 1999.

[3] R. Renesse, K. Birman, M. Hayden, A. Vaysburd and D. Karr, *Building Adaptive Systems Using Ensemble*. Cornell University Technical Report, TR97-1638, July 1997.

[4] M. Hiltunen and R. Schlichting, *The Cactus Approach to Building Configurable Middleware Services*. Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000), Nuremberg, Germany (October 2000).

[5] M. Jelasity and O. Babaoglu, *T-Man: Gossip-based overlay topology management*. In EngineeringSelf-Organising Systems: Third International Workshop (ESOA 2005), Revised Selected Papers.

[6] O. Babaoglu, *Gossiping in Bologna*. http://www.cs.cornell.edu/Courses/cs514/2007sp/UniBo%20Project/Leiden-Gossip.ppt.

[7] K. Birman, A. Abbadi, W. Dietrich, T. Joseph and T. Raeuchle, *An Overview of the ISIS Project*. IEEE Distributed Processing Technical Committee Newsletter. January 1985.

[8] N. Bhatti, M. Hiltunen, R. Schlichting and W. Chiu, *Coyote: A System for Constructing Fine-Grain Configurable Communication Services*. ACM Transactions on Computer Systems, November 1998.

[9] A. Ganesh, A.-M. Kermarrec and L. Massoulie, *SCAMP: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication*. In Proc. of the 3rd International workshop on Networked Group Communication, 2001.

[10] D. Agrawal, A. E. Abbadi and R. Steinke, *Epidemic algorithms in replicated databases*. In Proc. 16th ACM Symp. on Principles of Database Systems, 1997.

[11] R. van Renesse, Y. Minsky and M. Hayden, *A gossip-style failure-detection service*. In Proc. IFIP Intl. Conference on Distributed Systems Platform and Open Distributed Processing, 1998.

[12] I.Gupta, R. van Renesse and K.Birman, *Scalable fault-tolerant aggrgation in large process groups*. In Proc. Conf. on Dependable Systems and Networks, 2001.

[13] A. Demers, D. Greene, C. Hauser et al. *Epidemic algorithms for replicated database maintenance*. In Proc. of the sixth annual ACM Symposium on Principles of distributed computing, 1987.

[14] Z. Haas, J. Halpern and L. Li, *Gossip-based Ad-Hoc Routing*. Unpublished. http://citeseer.ist.psu.edu/article/haas02gossipbased.html

[15] M. Clarke, G. Blair, G. Coulson and N. Parlavantzasco *An efficient component model for the construction of adaptive middleware*. In Proc. of IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware). Germany, 2001.

[16] M. Hiltunen, F. Taiani and R. Schlichting, *Reflections on Aspects and Configurable Protocols*. The Fifth International Conference on Aspect-Oriented Software Development (AOSD.06), Bonn, Germany, March 20-24, 2006, pp.87-98 (12 p.).

[17] M. Jelasity, R. Guerraoui, A.-M. Kermarrec and M. Steen, *The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations*. Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, Toronto, Canada, 2004, pp. 79-98.

[18] C. Flores-Cortes, G. Blair and P. Grace, *A Multi-protocol Framework for Ad-Hoc Service Discovery*. In Proc. of the 4th International Workshop on on Middleware for Pervasive and Ad-Hoc Computing (MPAC '06), co-located with Middleware 2006, Melbourne, Australia, 2006.

[19] P. Grace, G. Coulson, G. Blair et al. *GRIDKIT: Pluggable Overlay Networks for Grid Computing*. In Proc.of International Symposium on Distributed Objects and Applications(DOA), Larnaca, Cyprus, 2004.