Facilitating Gossip Programming with the GossipKit Framework

Shen Lin, François Taïani, and Gordon S. Blair

Computing Department
Lancaster University, UK
{s.lin6,f.taiani,gordon}@comp.lancs.ac.uk

Abstract. Gossip protocols have been successfully applied 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.

Keywords: Gossip protocol, component framework, middleware, flexibility, event-driven architecture.

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 two 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) since gossip peers are selected at random and each node communicates with a limited number of peers, they offer natural load-balancing abilities. Because of these benefits, gossip-based protocols have been applied to a wide range of problems such as peer sampling [9,17], ad-hoc routing [14], reliable multicast[1,2], database replication [10], failure detection [11], and data aggregation [12].

In spite of this success, however, very few attempts have been made at developing gossip-based middleware architectures. T-Man [5] and Gossiping Framework [6] proposed by Kermarrec and Steen [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

R. Meier and S. Terzis (Eds.): DAIS 2008, LNCS 5053, pp. 238–252, 2008.

gossip-based applications. First, the common periodic gossip pattern they rely on only captures the features of proactive gossip protocols but does not support reactive gossip algorithms. Second, these frameworks tend to be monolithic, thus precluding a flexible and easily extensible architecture. Finally, these frameworks are not designed to 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 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 our current implementation, while an 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 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 frameworks are T-Man [5] and Gossiping Framework [6] proposed by Kermarrec and Steen [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. This inherently supports a large range of proactive gossip protocols such as peer sampling service, data aggregation, and topologic maintenance, which have all been implemented in such gossip frameworks.

However, the monolithic design of these frameworks makes them inadaptable 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 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 can be used 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 each new protocol from scratch. 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 above shortcomings, 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

GossipKit is based on three key design choices: (i) application-dependent interfaces; (ii) a common-interaction pattern, and (iii) an event-driven architecture. These choices result from a detailed analysis of a number of existing gossip-based

protocols. In the following, we discuss in turn each of our choices, and explain how they derive from this analysis.

3.1 Application-Dependent Interfaces

Gossip-based solutions have been proposed for a wide range of problems, and for each specific problem, external modules are expected to interact with the gossip protocol in very specific ways. For instance, a gossip-based routing protocol has to provide a way for external applications to trigger a route request to be gossipped, whilst a peer sampling service must instead expose the set of collected peers. There is no elegant way to map those fundamentally different services onto a unique common generic interface. Instead we have identified a *set* of generic but domain-specific interfaces that can each support a category of gossip protocols in a particular application domain (e.g. ad-hoc routing, or peer-sampling). This approach allows us to uncouple the varied semantics of gossip-based services from the unified implementation framework we have developed. We will revisit this topic in Section 4.1, where we will describe in more detail the mapping between these domain-specific interfaces and our underlying framework.

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 existing gossip frameworks [5,6], 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.

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

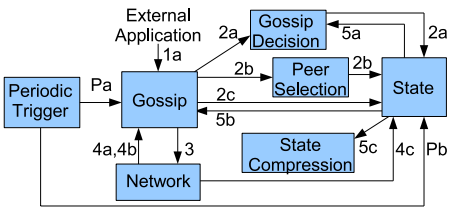

Fig. 1. Common Interaction Model

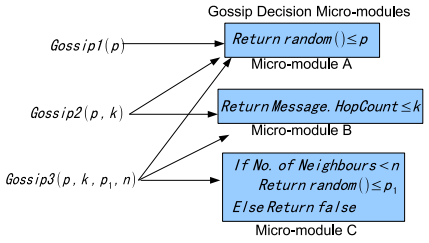

Fig. 2. Various Gossip Decision modules composed by micro-modules

(e.g. an ad-hoc routing protocol requesting a reactive gossip protocol such as [14] and [21] to gossip a route request). These two interactions are represented as (Pa) and (1a) 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 state (2a). For instance, a reactive gossip protocol may decide not to gossip the same message twice or forward the message with a given probability [9]. If a decision is made to forward the gossip message, the protocol instance will then select the peers it wishes to gossip with from its state (2b). Different policies exist for selecting peers. For instance, a subset of peers can be selected from the local state randomly or based on their lifetime [17]. Note that peer selection may be optional in our model: for instance the nodes of a wireless single-hop network always reach all their neighbours with a single radio broadcast, [20], gossip protocols operating in these environments therefore usually achieve gossiping behaviour through randomised gossip decisions. In addition to gossip decision and peer selection, 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 (e.g. a temperature reading) if it needs to send periodically its state (push-style gossip) or reply to a request of its state (pull-style gossip).

The third phase is gossip dissemination (3). It utilises the underlying network to send gossip messages to either the selected (e.g. in wired networks) or neighbouring (e.g. in MANETs) peers.

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: *i)* it might forward the message to peers that it knows (4a), thus repeating phase 2 (2a, 2b and 2c); *ii)* it might respond with its own state (4b), and again loop on phase 2); and *iii)* it might extract the remote state contained in the message, either merging [17] or comparing [2] the remote state with its own (4c). Besides this reactive behaviour, a gossip protocol may also update its own state periodically (Pb). For instance: a peer sampling protocol [17] may select peers by periodically their observed lifetime. Finally, a gossip protocol might invoke three different interactions during the state update process: 1) it might need to decide whether to merge the remote state with its local one (5a) based on certain probabilistic policies [9]; 2) it might compress the merged state (5b) to fit a predefined limit on the state size [2]; and 3) it might request for the missing information through the Gossip module (5c) after comparing the content in the remote state with its local one [1,2].

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 itself be a gossip-based peer sampling service protocol.

In practice, the modules in Fig. 1 are rather coarse-grained, and may vary widely between gossip protocols, making them hard to reuse. To maximise reuse, our framework therefore allows each module to be composed from finer-grained micro-modules, as shown on Fig. 2. More precisely, we have noticed that five modules (*Gossip*, *Peer Selection*, *Gossip Decision*, *State Compression*, and *State*)

can often be decomposed into finer-grained and reusable entities we have termed *micro-modules*. These micro-modules each implement a distinct algorithm, and can be combined to create more sophisticated behaviours. Fig. 2 shows for instance three gossip-decision policies used in a gossip-based ad-hoc routing protocol ($Gossip1(p)$, $Gossip2(p, k)$, and $Gossip3(p, k, p_1, n)$) [14].

$Gossip1$, $Gossip2$, and $Gossip3$ differ by how they decide whether to forward the received routing request message (i.e. they use different Gossip Decision modules): $Gossip1$ 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.

Rather than using separate implementations, these three different gossip decision strategies can be implemented by combining the three micro-modules shown on Fig. 2. $Gossip1$ can directly use micro-module A; $Gossip2$'s Gossip Decision module can be realised by combining with a Boolean OR the return values of Micro-modules A and B ; and $Gossip3$ can similarly be composed from micro-module A, B, and C. These different compositions are described in an XML configuration file that we will present in Section 6.1.

3.3 Event-Driven Architecture

To support the common interaction pattern we have just presented, we argue that any generic architecture should satisfy the following two criteria: First, it should facilitate the implementation of the various modules we have just described by making micro-modules easy to implement and configure. Second, Gossip protocols exist that we have not considered, and new ones will appear in the future, hence it requires extra modules and interactions beyond those we have identified, making extensibility a key requirement.

Both requirements can be fulfilled using an event-driven architecture. Traditional event-driven architectures such as Ensemble and Cactus allow flexible protocol configuration through bindings between event handlers and events. In such event-driven frameworks, our micro-modules (e.g. the Gossip Decision micro-modules in Fig. 2) can be viewed as event handlers that are bound to certain events. On the basis of these traditional event-driven architectures, GossipKit can be further improved to capture micro-module composition (e.g. the ones mentioned in Section 3.2) using extended event-bindings. For instance, to compose a 'Gossip Decision' module in GossipKit, several micro-modules can be bound to events raised by the 'Gossip' module (Fig. 1). The 'Gossip' module can then combine the values returned by each micro-module with a Boolean OR as part of the binding, and decide whether to forward the message.

Similarly, extensibility is addressed by using events to minimise explicit coupling between modules as argued in [8]. This allows our framework to be easily extended by plugging in new micro-modules (i.e. event handlers) and reconfiguring event bindings to support new interaction patterns.

4 GossipKit's Architectural Overview

The three design choices we presented in Section 3 have resulted in an architecture consisting 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.

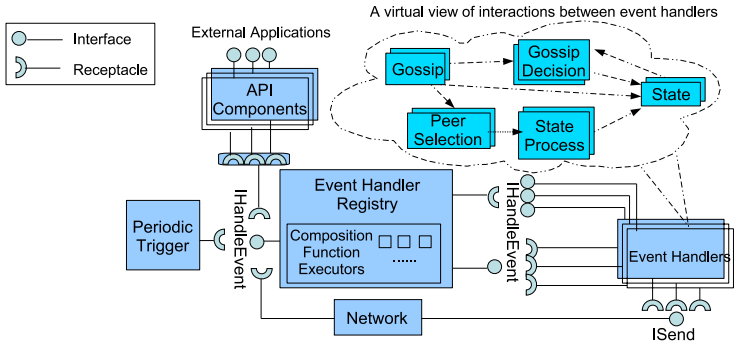

Fig. 3. GossipKit Architecture

4.1 API Components

API components uncouple the gossip protocols implemented by the framework from external applications. Each type of API component provides a generic interface to access a particular category of gossip protocols. API components also act as a bridge between their method-based interface and the events used by the framework. Fig. 4 for instance shows how the API component for the peer sampling service provides an IGetPeers interface to retrieve peer information from 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). The registry dispatches this event to the proper event handler (operation 3, see Section 4.3 below), which 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 returns the resulting peer sample to the external application through IGetPeers interface (operation 6).

4.2 Periodic Trigger Component

The periodic trigger component is only needed when the framework is used to support proactive gossip protocols. This component periodically dispatches

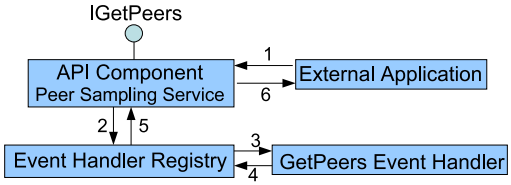

Fig. 4. Interaction of API Component with External Application

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 set at deployment time, and can be reconfigured dynamically.

4.3 Event Handler Registry

The event handler registry acts as a broker between event handlers and event producers (components that raise 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. To this aim, the registry maintains a table that records event handler IDs with the events they can handle. This table is populated each time an event handler's IHandleEvent interface is connected to the registry using the handler's meta-data. The event handler registry also provides an IHandleEvent interface to event producers to trigger the events.

Interestingly, the event handlers themselves can use the IHandleEvent interface to raise and delegate internal events to others handlers, thus providing a consistent event-based environment and facilitating interoperability between different gossip protocols.

Finally, the event handler registry can dynamically load composition functions to compile and interpret descriptions of micro-module composition, such as the ones mentioned in Section 3.2.

4.4 Event Handler Plugins

As mentioned in Section 3.3, our modules (i.e. Gossip, Peers_Selection, Gossip_Decision, State_Compression, and State in Fig. 1) can be further decomposed into finer-grained micro-modules. In our architecture, these micro-modules are implemented through a collection of event handler plugins (Fig. 3). These micro-modules are directly invoked by the event handler registry to handle events generated by the rest of the framework (including other micro-modules) using the extended bindings we've presented earlier. Micro-modules for the Gossip module have also access to network component to send messages (see below).

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, our framework can operate on heterogeneous transport layers such as UDP, TCP, or ad-hoc routing, or any virtual transport layers such component-based virtual overlays [19].

5 Implementation

GossipKit's prototype implementation[1] is based on the Java version of OpenCom [15], a lightweight, efficient and reflective component engine. 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 plain Java class. This class contains: (i) a header string, which identifies the event type used by the handler registry to find appropriate event handlers, (ii) a body containing data to be processed by event handlers, (iii) a source ID denoting the peer that generated the event, and (iv) a target ID that identifies the target peer the event should be routed to.

Our periodic trigger component features a basic yet efficient task scheduler that allows the coexistence of multiple gossip protocols working at different frequencies. Our scheduler uses a single thread shared for protocols, and thus significantly reduces resource utilisation on constrained systems. We will revisit this issue at Section 6.3 when we discuss the memory measurement of GossipKit.

6 Evaluation

We evaluated five key properties of GossipKit—(i) configurability, (ii) reusability, (iii) memory usage, (iv) extensibility, and (v) reconfigurability—by implementing three gossip protocols from two categories: the peer-sampling services SCAMP and PSS [9,17], and the reliable multicast 'Bimodal Multicast' [2]. To assess GossipKit's ability to support concurrent execution of multiple protocol instances, we also configured Bimodal Multicast to operate on SCAMP and PSS.

6.1 Configurability

In event-driven systems, manually configuring event bindings is often time-consuming. To ease this, GossipKit uses an XML-based configuration format (Fig. 5) that describes each protocol as a high-level component composition. This format uses the common interaction pattern we have identified earlier (Fig. 1) as a template that guides users through the selection process of interactions and module instances required to form a gossip-based protocol/application.

[1] Source code available at: `www.lancs.ac.uk/postgrad/lins6/sub/GossipKitWeb/GossipKit.html`

GossipKit's XML configuration format abstracts away the details of our event-driven architecture, and allows GossipKit to automatically map high-level protocol configurations to appropriate event generators and event handlers. GossipKit's configuration format contains the following key entities: 1) coexisting protocol instances are described using <protocol> elements; 2) the micromodules that make up each gossip protocol are described in <micromodule> elements, and can be parametrised individually using the <parameters> element; and 3) a dedicated non-XML syntax is used to describe textually compositional or recursive modules: for instance the Peer_Selection module for Bimodal Multicast is described as protocol(PSS) in Fig. 5 to indicate that PSS is used recursively to select peers; and the compositional GossipDecision module of *Gossip*3 in Fig. 2) would be described as micromodule(A OR B OR C).

From our experience, configuring a new protocol from existing elements takes approximately 15 minutes. For illustration, the remainder of this subsection shows how the PSS protocol can be configured to use push-style gossip and life-time based peer selection from existing events and micro-modules.

Before discussing PSS, we must first explain the various event types that label inter-module interactions in Fig. 6. As explained in Section 5, each event's type is encoded in a string-based header to help the event handler registry dispatch the event to the correct handlers. For instance, a State module can handle Get and Add events while a Gossip module can handle Gossip events. In addition to this base type, a header string can carry extra information to indicate the type of data either carried by the event, or that is to be retrieved from the state (i.e. our events are similar to *generics*). For instance, Get<PeerID> will instruct the State module to get a list of PeerIDs instead of the whole state. This mechanism is recursive, which allows for cascading events, such as when a Gossip module receives a Gossip<Add<PeerID>> event, and dynamically raises a Add<PeerID> event to be sent over the network.

```
<protocols>
    <protocol>
        <micromodules>
        <micromodule moduleID="TCP">
            <parameters>...</parameters>
        </micromodule>
            .....
        </micromodules>

        <interactions>
        <interaction>
            <source module="Gossip1"/>
            <target module="protocol(PSS)"/>
            <events>...</events>
        </interaction>
            .....
        </interactions>
    </protocol>
    <protocol>...</protocol>
</protocols>
```

Fig. 5. XML Config

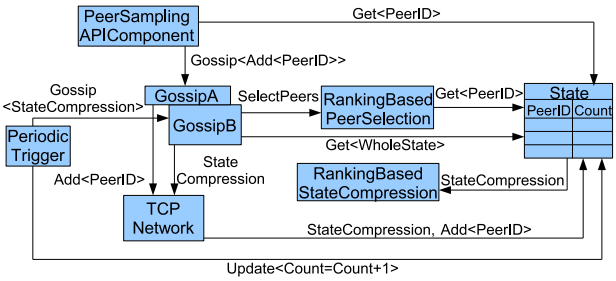

Fig. 6. Use case study: configuring the PSS protocol

In Fig. 6, the State is configured as a set of PeerIDs with associated lifetime counts. A local peer P joins the network by sending a Gossip<Add<PeerID>> event to an existing network peer Q, and retrieves its local peer sample with a Get<PeerID> event. On a join, GossipA is configured to forward the returned

content (i.e. in this case P's PeerID) to a given target (i.e. in this case Q) by sending the event Add<PeerID> on the network. When it receives this event, Q extracts P's PeerID from the event body and adds it to its state. The Periodic Trigger module dispatches two events periodically: 1) The first event increments the Count associated with each PeerID in State; while 2) the second triggers GossipB to push the content of the local state to selected peers. GossipB invokes the Ranking_Based_Peer_Selection micro-module to select peers based on their lifetime, and forward them its own state, obtained using Get<WholeState>. GossipB then sends this information within a StateCompression event to the selected peers. Each recipient then appends the received state to its own, before, the Ranking_Based_State_Compression micro-module compresses the size of the resulting state by discarding the PeerIDs entries with the oldest lifetime.

6.2 Reusability

We evaluated the reusability of GossipKit using a quantitative approach suggested in [18]. This approach measures the byte code size of the Java classes that make up different configurations of components. To evaluate the reused development effort, we initially considered to measure both the reused byte code size and the cyclomatic complexity [22], but as shown in Fig. 7, the byte code size and the cyclomatic complexity (measured using CyVis[2]) provide roughly the same indication of development effort. In the following we therefore limit ourselves to byte code measurements.

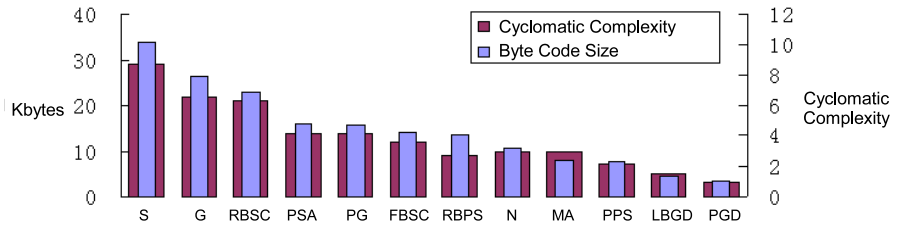

Fig. 7. Byte Code size and cyclomatic complexity provide the same measurements

In table(a) of Fig. 8, the columns under the protocol name SCAMP, PSS, and Bimodal Multicast list the number of each component type used for configuring these protocols. The highlighted rows show the components that were used more than once during configuration. These results show that most components have been frequently reused during the development of the three protocols. Furthermore, GossipKit does not only promote component reuse for developing gossip protocols that belong to the same category (SCAMP and PSS belong to the peer sampling category), but also for those belong to different categories (PSS and Bimodal Multicast). Finally, we compared the total effort of developing these

[2] `http://cyvis.sourceforge.net/`

Components (IDs) \ Config.		No. of Components Used			Framework size with 3 protocols (Kbytes)	Side by side size with 3 protocols (Kbytes)
		SCAMP	PSS	Bimodal Multicast		
Multicast APIComponent	(MA)	0	0	1	2.43	2.43
PeerSampling APIComponent	(PSA)	1	1	0	4.82	9.64
Gossip	(G)	2	2	3	7.88	39.4
LimitBased GossipDecision	(LBGD)	0	0	3	1.35	4.05
Probabilistic GossipDecision	(PGD)	2	0	0	1.00	2.00
PeriodicGossip	(PG)	1	1	1	4.68	14.04
Probabilistic PeerSelection	(PPS)	1	0	use PSS or SCAMP	2.32	2.32
RankingBased PeerSelection	(RBPS)	0	1	use PSS or SCAMP	4.00	4.00
Network	(N)	1	1	1	3.21	9.63
State	(S)	1	1	1	10.18	30.54
RankingBased StateCompression	(RBSC)	0	1	0	6.86	6.86
FilterBased StateCompression	(FBSC)	0	0	1	4.26	4.26
Total					52.99	129.17

Table (a) Reusability Measurement in Byte Code Size

Protocol	Memory size (bytes)		
	GossipKit	OpenCom	Java
PSS	14744	6160	15415
SCAMP	704	6600	24623
Bimodal Multicast	8216	8352	14786

Table (b) Runtime Memory Footprint Size Measurement

Fig. 8. Reusability and Memory Usage Measurements

three protocols in GossipKit (framework size) against the effort for developing each individual protocol without the support of GossipKit (side-by-side size). The result in table(a) of Fig. 8 shows that, overall, GossipKit helps save about 60% of development effort when implementing the three protocols.

6.3 Memory Usage

GossipKit aims to facilitate the development of a wide range of gossip protocols across heterogenous networks and devices. To assess GossipKit's suitability for mobile devices with strict memory constraints, we measured the dynamic memory footprint of the components that make up the protocol configurations at runtime, using the JProfiler[3] tool. The results in table(b) of Fig. 8 indicate that the configurations map well onto mobile devices, as minimum configurations of protocols in GossipKit require less than 100Kbytes. In addition, JProfiler shows the memory usage of the PeriodicGossip component that adopts a single thread implementation remains 16 bytes regardless to the number of concurrent protocol instances running in GossipKit, validating our choice of avoiding memory-intensive multi-threading mentioned in Section 5.

6.4 Extensibility

To assess GossipKit's extensibility, we used a case study to evaluate the effort required to add a new gossip-based protocol to the three existing ones. More precisely we developed a gossip-based number averaging protocol [6] based on the existing configuration for PSS. Fig. 9 shows that this new protocol can reuse most of PSS's modules and XML configuration file: One only needs to implement two extra modules (with development effort of 3.7 Kbytes measured in byte

[3] `http://www.ej-techonologies.com`

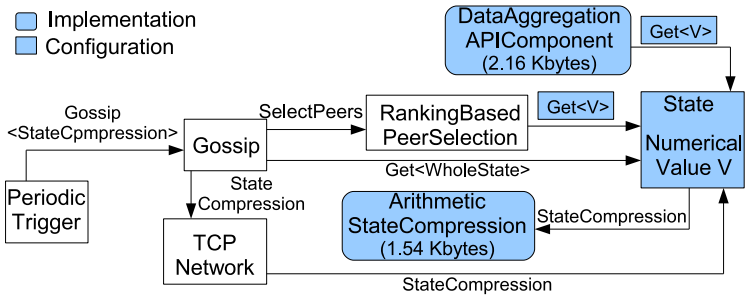

Fig. 9. Extending the PSS implementation into a number averaging protocol

code size) that are presented as shaded rectangles, to remove several redundant interactions and to change several configurations that are presented as shaded and rounded rectangles (the modifications on the configuration file only takes about 1 minute). Furthermore, we consider that it is less frequent for users to implement new components as the component collection expands because of GossipKit's support on code reuse. For instance, the newly implemented two modules will remain available for other data aggregation protocols and will not need to be re-implemented in the future.

6.5 Reconfigurability

GossipKit supports fine-grained reconfiguration to adapt to environment changes — different protocol behaviours can be achieved by replacing a single component. For instance, a peer sampling service with a life-time based peer selection can be replaced by a probabilistic peer selection module, and a particular network component can be replaced by different routing schemes. This form of component replacement relies on the mechanisms directly provided by OpenCOM. A 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 that aims to facilitate the development of configurable and reconfigurable middleware and supports multiple gossip protocols potentially operating in parallel under different types of networks. We have presented a 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 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 developing a domain specific visual language based on the existing XML-based configuration to further reduce the configuration effort and to guard users from potentially incorrect configurations. 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.

Acknowledgement

This work has been partially supported by the ESF MiNEMA programme.

References

1. Eugster, P., Guerraoui, R., et al.: Lightweight Probabilistic Broadcast. In: IEEE International Conference on Dependable Systems and Networks(DSN 2001) (2001)
2. Birman, K., Hayden, M., et al.: Bimodal multicast. TR99-1745, May 11 (1999)
3. Renesse, R., Birman, K., Hayden, M., et al.: Building Adaptive Systems Using Ensemble. Cornell University Technical Report (1997)
4. Hiltunen, M., Schlichting, R.: The Cactus Approach to Building Configurable Middleware Services. In: Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000), Nuremberg, Germany (October 2000)
5. Jelasity, M., Babaoglu, O.: T-Man: Gossip-based overlay topology management. In: EngineeringSelf-Organising Systems: 3rd International Workshop (2005)
6. Kermarrec, A., Steen, M.: Gossiping in Distributed Systems. In: Proc. of SIGOPS Operating System Review (2007)
7. Birman, K., Abbadi, A., Dietrich, W., et al.: An Overview of the ISIS Project. In: IEEE Distributed Processing Technical Committee Newsletter (January 1985)
8. Bhatti, N., Hiltunen, M., Schlichting, R., Chiu, W.: Coyote: A System for Constructing Fine-Grain Configurable Communication Services. ACM Transactions on Computer Systems (November 1998)
9. Ganesh, A., Kermarrec, A.-M., Massoulie, L.: 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. Agrawal, D., Abbadi, A.E., Steinke, R.: Epidemic algorithms in replicated databases. In: Proc. 16th ACM Symp. on Principles of Database Systems (1997)
11. van Renesse, R., Minsky, Y., Hayden, M.: A gossip-style failure-detection service. In: Proc. IFIP Intl. Conference on Distributed Systems Platform and Open Distributed Processing (1998)
12. Gupta, I., van Renesse, R., Birman, K.: Scalable fault-tolerant aggrgation in large process groups. In: Proc. Conf. on Dependable Systems and Networks (2001)
13. Demers, A., Greene, D., Hauser, C., et al.: Epidemic algorithms for replicated database maintenance. In: Proc. of the sixth annual ACM Symposium on Principles of distributed computing (1987)
14. Haas, Z., Halpern, J., Li, L.: Gossip-based Ad-Hoc Routing. IEEE/ACM Transactions on Networking (TON) (2006)

15. Clarke, M., Blair, G., Coulson, G., et al.: 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 (2001)
16. Hiltunen, M., Taiani, F., Schlichting, R.: Reflections on Aspects and Configurable Protocols. In: The 5th Int. Conf. on Aspect Oriented Software Development (2006)
17. Jelasity, M., Guerraoui, R., Kermarrec, A., et al.: The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations. In: Proc. of the 5th ACM/IFIP/USENIX international conference on Middleware (2004)
18. Flores-Cortes, C., Blair, G., Grace, P.: 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, Australia (2006)
19. Grace, P., Coulson, G., Blair, G., et al.: GRIDKIT: Pluggable Overlay Networks for Grid Computing. In: Proc.of International Symposium on Distributed Objects and Applications(DOA), Larnaca, Cyprus (2004)
20. Friedman, R., Gavidia, D., Rodirgues, L., et al.: Gossiping on MANETs: the Beauty and the Beast. ACM Operating Systems Review (2007)
21. Hou, X., Tipper, D.: Gossip-based sleep protocol (GSP) for energy efficient routing in wireless ad hoc networks. In: Proceedings of Wireless Communications and Networking Conference (2004)
22. McCabe: A Complexity Measure. IEEE Transactions on SE (1976)