GOSSIPKIT: A Unified Component Framework for Gossip

François Taïani, Shen Lin, Gordon S. Blair
E-mail: f.taiani@computer.org, shen.lin@outlook.com, g.blair@lancaster.ac.uk

Abstract—Although the principles of gossip protocols are relatively easy to grasp, their variety can make their design and evaluation highly time consuming. This problem is compounded by the lack of a unified programming framework for gossip, which means developers cannot easily reuse, compose, or adapt existing solutions to fit their needs, and have limited opportunities to share knowledge and ideas. In this paper, we consider how *component frameworks*, which have been widely applied to implement middleware solutions, can facilitate the development of gossip-based systems in a way that is both *generic* and *simple*. We show how such an approach can maximise code reuse, simplify the implementation of gossip protocols, and facilitate dynamic evolution and re-deployment.

Index Terms—distributed systems, components, frameworks, protocols

1 INTRODUCTION

Gossip protocols[1] have attracted a considerable amount of attention over the last decade. Their natural robustness, scalability, and self-stabilisation have made them particularly well adapted to the needs of data-centres [2], [3], [4], wireless sensor and mobile ad-hoc networks [5], [6], [7], and more recently social networks [8], [9], both in fixed IP-based networks [4], [2], [10], [11], [12] and wireless environments [5], [13], [14].

Gossip protocols use randomised communication to distribute information over a network in the same way a rumour is gossiped amongst people. This causes most gossip protocols to follow a bi-modal behaviour similar to that of disease epidemics [2]: as soon as the probability of propagation meets some minimum threshold, a gossiped message will be received by all or almost all nodes with a very high probability. This phenomenon makes gossip protocols highly scalable, self-organising, and resilient to failures.

Although the principles of gossip protocols are relatively easy to grasp, their variety—in terms of provided services, targeted properties, and assumptions made on their environment—can make their design, implementation, and evaluation highly time consuming. In particular, the lack of a unified programming framework for gossip protocols means that developers cannot easily reuse, compose, and adapt existing solutions to fit their needs, which limits opportunities for knowledge sharing and cross-pollination.

In this paper, we consider how *component frameworks* [15], [16] can help address this gap. *Component frameworks* are a modular programming approach that has been successfully applied to many areas of distributed systems [17], [18], [19]. They allow developers to assemble systems from reusable *software components* according to domain-specific rules. Software components are encapsulated software entities that explicitly expose their operational dependencies, typically in the form of interfaces and receptacles (i.e. provided and required services). They thus encourage a compositional approach to system construction that fosters modularity, reuse, and configurability. They also facilitate the development of dynamically adaptive systems: the use of explicit interfaces and receptacles make it simple to reason about dependencies, while dynamic bindings provide a simple mechanism to update a system at runtime.

To demonstrate how component frameworks can support the development of gossip-based systems, this paper introduces a unified programming framework for gossip protocols called GOSSIPKIT. GOSSIPKIT offers a component-based architecture that promotes code reuse and simplifies the implementation of a wide range of gossip protocols. GOSSIPKIT also allows multiple protocol instances to be dynamically loaded and reconfigured, operate concurrently, and collaborate with each other in order to achieve more sophisticated operations.

The contributions of this paper are threefold. *First*, after reviewing related work (Sec. 2), we present a survey of existing gossip protocols, and identify a set of core design dimensions, strategies, and patterns that underpin the design of most gossip protocols (Sec. 3). *Second*, we propose GOSSIPKIT, a generic component-based framework that captures those recurring elements and seeks to unify the construction of gossip-based systems (Sec. 4). *Finally*, we evaluate GOSSIPKIT and show that it considerably simplifies the implementation of gossip protocols, while fostering reuse, and providing the benefits of component frameworks in terms of configurability and dynamic adaptation (Sec. 5). We end by offering some concluding remarks (Sec. 6).

2 RELATED WORK & PROBLEM STATEMENT

We first present the tenets of gossip protocols (Sec. 2.1), protocol kernels (Sec. 2.2), and component frameworks (Sec. 2.3).

- *F. Taïani is currently with the University of Rennes 1 / IRISA (France); S. Lin is with SAP Labs (China); G. S. Blair is with the University of Lancaster (UK). F. Taïani and S. Lin were with the University of Lancaster (UK) while the work reported in this paper was conducted.*

- *A preliminary version of this paper was published in [1].*

1. Also known as 'epidemic' protocols.

We then review earlier attempts to systematise the programming of gossip protocols (Sec. 2.4), and finally discuss the challenges inherent to the application of components to gossip (Sec. 2.5).

2.1 Gossip protocols

Gossip protocols take inspiration from disease epidemics and rumour dissemination to implement distributed computer algorithms. Due to their wide variety [20], [8], [14], [21], [22], [23], [24], [2], proposing a definitive definition of gossip protocols remains difficult. In this work we follow earlier authors [25], [26] and consider that gossip protocols are *round-based*, *message-passing*, *decentralised* computer algorithms, in which (i) *stateful nodes* exchange information with *a few other nodes* (compared to the overall size of the system) during every round; and (ii) this exchange is *probabilistic*. Contrary to some authors [25], [26], we do not assume that rounds are necessarily periodic. They might in our model be triggered by sporadic events. (We revisit this point in Sec. 3.)

The nature of the state stored on each node, the type of data being exchanged, and the stochastic rules by which nodes interact, all contribute to determining which service (e.g. broadcast, topology construction, system partitioning) is provided by a protocol. For instance, a robust and highly scalable broadcast algorithm can be obtained by having nodes store a history of the messages seen so far (local state), and retransmit each new message (exchanged data) to k randomly selected other nodes (interaction rule) [27]. Conversely, a family of either peer sampling [28], [20] or topology construction gossip protocols [11], [8] can be constructed when each node uses a small list of other nodes (the node's *view*) as its local state, and updates this list using its neighbours' lists.

Gossip protocols offer four key advantages over more traditional systems: 1) they are particularly scalable; 2) they are naturally robust to failures; 3) they are reasonably efficient; and 4) they can often be configured to fit varying needs by changing a few central parameters (e.g. fanout). As a result, they have been applied to a wide range of problems such as peer sampling [29], [20], wireless routing and broadcast [5], [6], [7], reliable multicast [30], [23], database replication [3], failure detection [24], and data aggregation [31].

Although the basic intuitions behind gossip protocols are easy to grasp, the power and complexity of the approach comes from the potentially infinite ways in which its constitutive ingredients (local state, data exchange, and stochastic interactions) can be combined. Individual protocols differ in how they trigger exchanges (in periodic or reactive rounds); in the type of state each node maintains (a measurement, a list of neighbours, a dictionary); in the stochastic mechanisms that drive data exchanges (biased, unbiased); in the information that nodes exchange (their whole state, or part of it); and in the mechanisms that nodes use to update their local state (e.g. ranking, shuffling, concatenation). Some protocols might also be composite: for instance a gossip-based broadcast protocol might rely on a peer-sampling gossip protocol to build and maintain a neighbourhood of other nodes [32], [33].

2.2 Protocol kernels

The approach we take in this paper to systematise the development of gossip protocols builds on a long tradition of *protocol kernels*, which seek to facilitate the development of a large range of distributed protocols from fine-grained reusable entities termed *micro-protocols*. Prominent examples include Ensemble [34], Cactus [35] and Appia [36], and their predecessors Isis [37], Horus [38] and Coyote [39]. In these environments, a distributed service (e.g. a leader-election protocol) is viewed as a composition of several functional properties (e.g. reliability, flow control, and ordering) encapsulated in micro-protocols. Micro-protocols generally consist of a collection of event handlers, whose interactions obey predefined rules (i.e. layers, types) imposed by the kernel. Ensemble and Horus for instance impose a purely layered architecture. Cactus by contrast relies on a two-level composition model, with micro-protocols freely bound using events to form (macro)-protocols, which in turn may be layered to realise a full system.

2.3 Component frameworks

Micro-protocols can be seen as the forerunners of *component frameworks* [15] applied to distributed protocols. Component frameworks are a modular programming approach that allows developers to assemble systems from reusable *software components*. Software components extend the notion of object orientation by introducing explicit dependencies between provided and required interfaces [16]. Component frameworks add rules and constraints on how software components might be assembled in order to capture the domain knowledge of a particular area [15]. As such, they encourage a compositional approach to system construction that fosters modularity, reuse, and configurability. They also facilitate the development of dynamically adaptive systems: knowledge about provided and required interfaces allows a reconfiguration engine to reason about dependencies, while dynamic bindings provide a simple mechanism to update a system at runtime.

These benefits have made components and component frameworks a particularly popular approach to develop distributed platforms. They have been successfully applied both in the industry (Enterprise Java Beans (EJB), the Service Component Architecture (SCA), the CORBA Component Model (CCM), .Net, and the OSGi Remote Services Specification), and in middleware research, giving rise to *lightweight component technologies* with reflective capabilities (OpenCom [40], and Fractal [41]) and their associated middleware frameworks (GridKit [17], RAPIDWare [18], FraSCAti [19]).

The work we present is particularly related to low-level component frameworks developed for embedded systems. The resulting frameworks are usually fine-grained, low-overhead, compact (a few Kbytes) and highly configurable (with most components typically optional). One well-known example is nesC, the C-derived language underlying the TinyOS operating systems for Wireless Sensor Networks [44]. Quite crucially, nesC configurations are static and cannot change at run-time. Successors to nesC, such as the LooCi component model [45] or our own OpenCom [40] have sought to alleviate this limitation, and allow for dynamic architectures on constrained

Approach	Nature	Composable By Assembly	Fine Granularity	Decomposition Unit	Prototype
Kermarrec & Steen [10], [11]	thread pattern			threads	
Eugster, Felber & Le Fessant [42]	API		✓	functions	
B^2 [25]	component framework	✓		component	
GCP [43]	annotation model		✓	annotations	✓

TABLE 1: Existing approaches to gossip programming

devices (a TelosB mote for instance) [46].

These lightweight technologies use very few resources (less than a few Kbytes per component in some instances), and are thus well adapted to construct low-level system software. Examples include wireless sensor networks [44], [46], [45], environmental sensing [47] and embedded fault-tolerance [48]. The use of components in such systems provides in turn a systematic approach to reason about their design, reuse, and dependencies in a clear, principled, and intuitive manner [40].

2.4 Gossip programming

Commenting on the shared foundations of gossip protocols, a number of authors have sought to propose general approaches to their design [42], [10], composition [25], and implementation [43]. Table 1 provides an overview of these earlier attempts, and compares them in terms of *composability*, *granularity*, *level of decomposition*, and whether they have been implemented (*prototype*).

With one notable exception [43], all these approaches have, to the best of our knowledge, remained purely conceptual. Furthermore none of these approaches supports a model of fine-grained elements that can be composed by assembly, as we do: B^2 [25] uses components, but considers individual protocols as monolithic black boxes. GCP [43] and the model of Eugster, Felber & Le Fessant [42] rely on a fine-grained decomposition, but in a form (functions [42] or annotations [43]), that does not lend itself to composition by assembly.

These two properties (fine granularity & composition by assembly) are two key contributions of our work. By decomposing protocols in fine-grained elements, we can deliver high levels of reuse (Sec. 5.2). By providing composition by assembly, we naturally support the dynamic reconfigurations associated with component frameworks (Sec. 5.5). In the following, we revisit the approaches of Table 1 in the light of these two properties, and contrast them with our approach.

Kermarrec and Steen [10] have observed that periodic pull-push gossip protocols can be implemented using two concurrent threads (also used in [11]), one *active* and one *passive*. The active thread periodically pushes the local state S_P to a randomly selected peer Q or pulls Q's local state S_Q. The passive thread replies to push or pull messages from other peers. This decomposition captures the distributed concurrency inherent to message passing systems, and is thus more a programming pattern than a programming model. It does not aim in particular to provide any reusable software block.

Eugster, Felber and Le Fessant [42] have extended this pattern and proposed a set of three fundamental pseudo code Application Programming Interfaces (APIs) to capture the recurring design dimensions of gossip protocols in terms of *randomness*, *neighbourhood*, and *communication*. Totalling seven functions, these APIs are concise: The one for neighbourhood management for instance allows one to retrieve, add, and remove a node's neighbours. Remaining conceptual, this approach requires developers to write traditional imperative code, and so does not lend itself to the kind of composition by assembly we advocate in this paper.

At a coarser level, Rivière, Baldoni, Li and Pereira [25] have proposed a *conceptual design framework* for gossip based on reusable building blocks (B^2). Although purely conceptual, these building blocks are closely related to software components, and aim to capture the input and output of individual gossip protocols. The B^2 approach focuses however on the composition of several protocols into a larger system, rather than on the implementation of individual protocols, as we do. Individual protocols are treated as monolithic black boxes, in stark contrast to our work.

Finally, Princehouse and Birman [43] have developed a code partitioning technique to help realise and analyse gossip based systems. Their approach, termed *Gossip Code Partitioning* (GCP), uses a high-level model of gossip interactions based on a functional representation. This high-level model is then automatically partitioned into code executing on individual nodes using plain Java code, Java annotations, reflection, program analysis (slicing) and byte-code rewriting. Like, GOSSIPKIT, GCP [43] seeks to decompose gossip protocols into their fine-grained constituting elements. It does not however focus on composition by assembly, as we do. As a result it does not by itself provide the kind of dynamic adaptation capabilities associated with component frameworks.

2.5 Componentising gossip

The approaches we have just presented all fail to provide a concrete set of fine-grained reusable entities which can be assembled to produce a large range of gossip protocols. Partitioning a family of algorithms (gossip protocols) into a component framework is, however, inherently challenging, and remains as much a craft as a science. Ideally the resulting framework should be both *simple*, and *generic*. These two aims unfortunately tend to oppose each other: A design that is too simple might exclude protocols falling outside its main design philosophy. Conversely, a highly generic framework might incur much complexity, and require a large effort of configuration to implement even simple instances.

In the rest of this paper, we aim to demonstrate how these two aims (*simplicity* and *genericity*) can be reconciled in the case of gossip protocols by founding our work on a systematic survey of a representative set of gossip protocols (Sec. 3), before moving on to present our design in more detail (Sec. 4).

3 Surveying Gossip Design Choices

Our survey covers a representative set of 33 gossip protocols (Table 2). In the following, we document the common features and variation points we have observed in this set in terms of design dimensions (Sec. 3.1), and summarise our findings as a set of common design patterns for gossip (Sec. 3.2).

3.1 Underlying design dimensions

Like many distributed algorithms, designing a gossip protocol requires one to make decisions about both *data* (which data to store, which data to exchange, in which data structures, using which update strategies), and *communication* (when to exchange data, in which direction, according to which stochastic patterns) [10], [42]. In our survey, we found that the data needs of most gossip protocols could be captured by a simple and generic storage schema, without much need for further decomposition. (We come back to this point in Section 4.4 when we discuss the detail of GossipKit.)

By contrast, we found that the communication of most gossip protocols could be further decomposed along three subdimensions mirroring the key stages of a gossip round: the *communication trigger*, the *style of randomisation*, and the *direction of data-flow* (6 middle columns of Table 2).

3.1.1 Communication Trigger

The Communication Trigger dimension captures how the rounds of a gossip protocol are initiated. A gossip round is a sequence of operations each node repeatedly executes as part of the protocol. A round can be *periodic* [20] or *reactive*, in which case it is triggered by external events [28], such as a new sensor reading, or the reception of a gossip message.

Periodic rounds effectively avoid possible traffic congestion, by distributing the sending times of individual nodes evenly within the interval of a round. In contrast, reactive rounds tend to propagate new information more rapidly, but generate a large amount of network traffic within a short period, which might cause congestion.

3.1.2 Style of Randomisation

The Style of Randomisation of a protocol captures the nature of the stochastic rules that govern its communication. We have found these rules to be either *one-to-one* or *one-to-many*.

With the *one-to-one* strategy, nodes explicitly select the peers with which to interact during each round. This strategy is predominant in fixed point-to-point networks. In this case, a few peers are typically drawn *uniformly* among the population of other peers (possibly relying on an appropriate peer sampling service). This uniform approach tends to optimise the convergence speed of the overall system to a stable state (e.g. a target topology, a coverage of all nodes with a broadcast message).

In some gossip protocols, however, the random selection of peers is not uniform, but *biased* according to specific criteria. For instance, directional gossip [49] takes into account the topology of a wide area network, and selects with a higher probability remote peers (e.g. nodes in different local networks) to accelerate the distribution of information. Similarly, Probabilistic Multicast [21] conditions the propagation of events on their properties to narrow-down the propagation to interested nodes.

The *one-to-many* strategy is preferred in gossip protocols operating in wireless environments such as Wireless Sensor Networks (WSNs), and Mobile Ad Hoc Networks (MANETs). Because one-hop wireless broadcasts tend to reach most nodes within a broadcaster's range, broadcast operations themselves are made stochastic rather than the selection of recipients. Furthermore, the decision to broadcast is usually based on parameters that are closely associated with wireless networks, e.g. node density, hop-count [5], energy [7] or traffic patterns [6].

3.1.3 Direction of Data Flows

Gossip protocols finally rely on three basic styles of data flows: *push-pull*, *push*, and *pull*. *Push-pull* propagates data both ways when two nodes interact, thus fostering the rapid convergence of the system to a desirable state. *Push-pull* can also be used to disseminate *digests* of the available data (*push*), and only trigger data transfer (*pull*) as needed. (See Sec. 3.2.3.)

In a *push* style gossip, each node sends its data to some random peers but does not require any reply from these peers. As a result, push style gossip uses only half as many messages per round as push-pull exchanges, but requires longer to converge when used for self-organisation (e.g. topology construction) of aggregation [11], [8]. Push style gossip works well when disseminating information, however, as there is in this case no need for recipients to reply to senders.

Finally, in a *pull* style gossip, a node queries some random peers for its data. Pulling ensures that data is only transferred when needed. It helps reduce network traffic when the size of the data to be gossiped is particularly large [67], [54], [64].

3.2 Key Patterns

The three design dimensions just presented, and the strategies they call for, usually appear in four common combinations, which we have termed *gossip patterns* (first column of Table 2). In the following, we review each pattern, discussing concrete illustrative examples as we go along.

3.2.1 Pattern P1: Punctual Dissemination

Punctual Dissemination combines a reactive trigger with a push data flow to propagate information on fixed networks, either alone or in combination with other types of gossip mechanisms. The protocol is triggered either when a node has new information to send, or when it receives a new message from another peer. When this happens, the receiver immediately resends the message to some randomly selected nodes.

Example: SCAMP [28] is a peer-membership protocol that uses punctual dissemination to balance the network connectivity so that each node maintains a view of $\log(N)$ evenly distributed random nodes. On receipt of a join request from

		Design Dimensions & Strategies						
		Trigger		*Data Flow*		*Randomisation*		
Pattern	**Subpattern**	**Reactive**	**Periodic**	**Push**	**Pull**	**1-to-1**	**1-to-m**	**Protocol Examples**
P1: *Punctual Dissemination*		✓		✓		✓		**SCAMP** [28], Directional Gossip [49], PlumTree [50], Unstructured Epidemic Multicast [51], Spatial Gossip [22]
P2: *Continuous Dissemination*	**a.** *Forward*		✓	✓		✓		G-SDP [52], G-FDS [24], RDG [14], GSGC [53], NEEM [54], [55], Gravitational Gossip [56], Hierarchical Gossip [27], lpcast [30], Probabilistic Multicast (multicast) [21]
	b. *Polling*		✓		✓	✓		Anonymous Gossip [57], RDG [14], Probabilistic Multicast (membership) [21]
	c. *Pairwise*		✓	✓	✓	✓		Cyclon [58], HyParView (passive views) [59], **RPS** [20], Topology Construction (T-Man [11], Vicinity [8], T-Chord [60], **T-Simple** (Sec. 5.1)), **Averaging** [12], **Ordered Slicing** [4], Newscast [61], Araneola [62], Anti Entropy (push/pull) [2]
P3: *Lazy Dissemination*	**a.** *Continuous*		✓	✓	✓	✓		**Bimodal Multicast (Anti Entropy)** [23], NEEM [54], [55], lpcast [30], TAG [63]
	b. *Punctual*	✓		✓	✓	✓		K-Walker [64], PlumTree [50], Unstructured Epidemic Multicast [51]
P4: *Broadcast*	**a.** *Explicit*	✓		✓			✓	Smart Gossip [65], Polarized Gossip [66], **Gossip**$_{1,2,3}$ [5]
	b. *Sleep based*	✓		✓			✓	GSP [6], T-GSP [7]

TABLE 2: Gossip design dimensions, strategies & patterns

node i, a SCAMP node n forwards the request to all the nodes in its view. On receipt of a forwarded request, a node j adds node i to its view with probability p, and otherwise forwards i's join request to a random node in its view. This propagation mechanism helps in turn balance the random graph every time a node joins the network.

3.2.2 Pattern P2: Continuous Dissemination

The *Continuous Dissemination* pattern combines a periodic trigger with one-to-one randomisation, and comes in three subpatterns depending on the direction of data flow: *Forward*, *Polling*, and *Pairwise*. During each round, each node selects a number of random peers, and then disseminates its information to these peers (i.e. push), requests information from them (i.e. pull), or both (i.e. push-pull). Gossip algorithms based on this pattern are often used to achieve convergence of some global properties (e.g. to achieve a particularly topology [11], [8], [62] or partitioning [4]) or to aggregate data (e.g. averaging) on fixed networks [12].

Example: The averaging [12] protocol uses periodic pairwise (pull-push) exchanges to estimate the average of a value held by each node, e.g. a temperature. In every round, each node n selects a random peer i, to which it sends its current value, while i does the same. n and i then update their value to be the average of v_n and v_i. As the protocol progresses, the values stored on individual nodes gradually converge to a global average.

Similarly, Araneola [62] uses this pattern to construct a balanced random overlay in which all nodes maintain the same out- and in-degree. Araneola combines a periodic trigger with a local condition on the state of nodes: nodes only gossip when their degree diverges from the target value (k or $k+1$).

3.2.3 Pattern P3: Lazy Dissemination

The *Lazy Dissemination* pattern uses the same push-pull strategy as the sub-pattern *Pairwise* of *Continuous Dissemination*, but employs the push and pull exchanges for two different and complementary aims. In this pattern, nodes do not send their data directly to other nodes, but disseminate instead digests of the data they hold to some randomly selected peers in each gossip round. When receiving a digest, a node queries the actual data if it is interested in the advertised content. Lazy Dissemination is often used to recover lost messages and implement reliable multi-cast protocols, and appears in both periodic and reactive protocol (*Continuous* and *Punctual* sub-patterns respectively).

Example: The Anti-Entropy protocol of Bimodal Multicast (also known as *pbcast*) [23] uses lazy dissemination to repair message losses in unreliable multicast systems.

3.2.4 Pattern P4: Broadcast

The fourth and final pattern, *Broadcast*, is similar to *Punctual Dissemination* (Pattern P1), but uses one-to-many randomisation to propagate information in a mobile or wireless sensor network (MANETs and WSNs). In the first sub-pattern (*Explicit*), a node i re-broadcasts new incoming messages to all nodes in its range with a certain probability. This probability might itself be dependent on contextual parameters (number of neighbours, observed retransmissions) [5], [65].

In a second sub-pattern (*Sleep-based*), a node uses sleep as a probabilistic communication control, rather than explicitly deciding on each broadcast. More precisely, only nodes that are awake forward messages in this pattern, while nodes enter sleep randomly for a give period T with a probability p. T and p might be fixed or themselves depend on additional contextual parameters. This pattern is primarily used in energy-constrained networks to save energy.

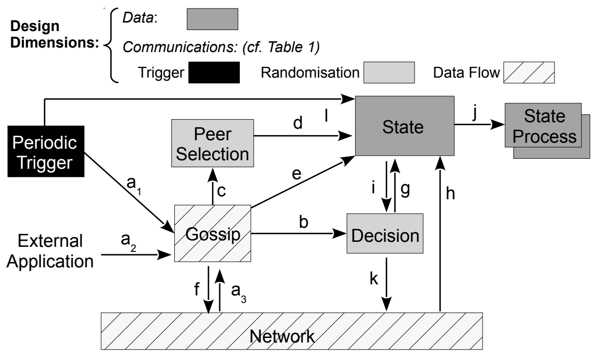


Fig. 1: GOSSIPKIT's architecture

Example: $Gossip2$ [5] is a wireless broadcast protocol designed for MANETs. The gossip decision of $Gossip2$ is based on four parameters: p_1, k, p_2, and n. To prevent messages from dying early, $Gossip2$ forwards requests with probability 1 during their first k hops. Then, nodes that have more than n neighbours gossip with a default probability p_1. To improve the delivery rate in sparse networks, nodes that have less than n neighbours gossip with a boosted probability $p_2 > p_1$.

3.3 Summary

The four patterns just presented (Sec. 3.2) capture the recurring combinations in which the design dimensions and strategies discussed in Section 3.1 are routinely combined in the 33 protocols we have analysed. These patterns highlight both the diversity of existing gossip protocols, and the recurring overlap between the mechanisms they use. This double observation hints at the potential benefits of component frameworks for the realisation of gossip protocols in a manner that is both *generic* and *simple*. These are the topics we turn to in the next section, where we present GOSSIPKIT, the component framework we have developed based on the analysis just presented.

4 A COMPONENT FRAMEWORK FOR GOSSIP

Genericity and simplicity are traditionally at odds in component frameworks (Sec. 2). To achieve both properties in GOSSIPKIT, we made two design choices: that of *fine-grained components*, to maximise the potential reusability of individual component implementations, and that of a *rich event-based* interaction model, to simplify component interactions, while maintaining some structure in our handling of events.

4.1 GOSSIPKIT's architecture

GOSSIPKIT involves seven *component roles*[2] that work together to realise the steps of a gossip round. To realise a concrete protocol, each of these roles must be instantiated with a component implementation either taken from a pool of components or specifically realised for this protocol (more on this below). In Figure 1, these roles are shown as rectangles, and their interactions as arrows. Arrow directions indicate which component initiates an interaction, and arrow labels show in which sequence these interactions typically occur. Component roles are shaded according to the *design dimension* they address, namely *Data* (Sec. 3.1), *Communication Trigger* (Sec. 3.1.1), *Style of Randomisation* (Sec. 3.1.2), and *Direction of Data Flows* (Sec. 3.1.3). The last three dimensions underpin the analysis we presented in Section 3, and correspond to the middle columns of Table 2.

In terms of roles, the Gossip component orchestrates the execution of each round. The Periodic Trigger component is optional and when present periodically triggers rounds and background work. The Peer Selection and Decision components, both also optional, implement respectively the *one-to-one* and *one-to-many* randomisation strategies of Table 2. Finally the State component stores the node's local state (which we detail further in Section 4.4), and the State Process components provide the state update mechanisms required by individual protocols.

Which component implementations are selected to fulfil the above roles determine which strategies (Table 2) a protocol uses. There is however no one-to-one relationship between component implementations and strategies: For instance, the default component library of GOSSIPKIT provides one single generic implementation of the Gossip role, which can be configured to implement different data flows (Sec. 4.4.2), but three implementations of the Peer Selection role.

In the following we detail the sequence of interactions captured by GOSSIPKIT's architecture (labels $\mathbf{a}_1$ to **l** in the figure), before presenting our event-based interaction model (Sec. 4.3), and finally discussing the workings of the State and Gossip components in more detail (Sec. 4.4).

2. When the distinction is clear in the following, we use the word *component* to mean both *component instance* and *component role*.

4.2 Sequence of interaction

In GOSSIPKIT, a gossip round might be triggered periodically ($\mathbf{a}_1$), as in RPS [20], or started in reaction to an application event ($\mathbf{a}_2$) or to an incoming gossip message ($\mathbf{a}_3$), as in reactive routing protocols [5], [68] (Sec. 3.1.1). The rest of the gossip round is then orchestrated by the Gossip component. First, a decision might be made whether to gossip at all (**b**), typically in wireless networks to support the one-to-many randomisation strategy (Sec. 3.1.2). This decision might be purely stochastic, or take into account additional inputs such as the current state (e.g. the number of neighbours [29], label **g**) or current networking conditions (e.g. traffic [6], label **k**).

If the decision is positive, a subset of neighbours is selected for communication from the list of neighbours (the *node's view*) stored in the State component (**c** & **d**, optional for wireless radio broadcast); a gossip message is constructed; and finally the message is disseminated (**f**). How the message is constructed depends on the type of protocol. In a periodic gossip pattern, the key data is typically maintained by the gossip protocol itself (e.g. a list of neighbours [11], [20], or database updates [2]), and the content of the message is extracted from the node's State component (**e**). By contrast, reactive protocols often receive their data (e.g. a message to be broadcast [5], [14], [13]) from some external source together with the protocol's trigger ($\mathbf{a}_2$).

On receiving a gossip message ($\mathbf{a}_3$), a node might directly update its internal state (e.g. merging neighbourhood lists [20],

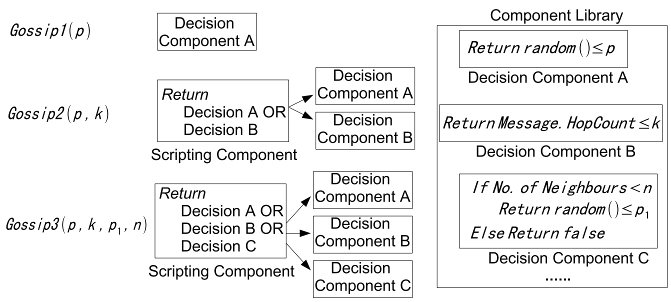


Fig. 2: Composite realisation of Decision components [5]

Attribute	Description
Event Type	Type of event
Event Source	Component that raised the event.
Protocol ID	protocol instance in which the event was raised
Sending Node	Identifier of the node that sends the event
Receiving Node	The node a remote event is sent to. Left blank when using wireless broadcasts.
Nested Events	An optional list of nested events
Payload	The data carried by the event

TABLE 3: Key Attributes of a GOSSIPKIT event

[8], [11]) using the message's content (**h** and **j**), possibly on the condition of a probabilistic decision (**i**). If the protocol is reactive (Sec. 3.1.1), the received message might also trigger another round of gossip. Alternatively, in case of pull style gossip (Sec. 3.1.3), the received message might cause the recipient to respond directly to the sender (**f**) using information stored in the State component (**e**). (We return to this latter aspect in Section 4.4 when we discuss nested events.) Finally, interleaved with gossip rounds, a protocol might also perform regular updates on the local node state (label **l**), for instance to keep track of time, or perform out-of-band bookkeeping operations such as garbage collection, or data optimisation.

To increase opportunities for reuse, each of the roles shown in Figure 1 can be implemented as a composite component, made of smaller components. In practice, we have found this possibility useful for two component roles: Peer Selection and Decision. Many gossip protocols [11], [8] use a peer-sampling service to select peers to gossip to, and this peer-sampling service is itself realised as a gossip protocol (e.g. RPS [20]). We capture this situation in our framework by recursively instantiating the Peer Selection role as a gossip protocol that itself follows the architecture of Figure 1.

Similarly, some gossip-protocols use composite criteria to decide whether to gossip. In such case, we assemble their Decision component from a set of smaller components. Figure 2 shows for example how the Decision component of a family of gossip-based ad-hoc routing protocols [5] can be realised from three basic micro-components [1].

4.3 Rich and uniform event interactions

The sequence of interactions we have just described is implemented in GOSSIPKIT using events, following in that respect the choice of earlier configurable communication platforms [35], [34], [39]. GOSSIPKIT uses *rich events* that carry a number of contextual parameters (e.g. protocol ID, event source, data payload). These rich events also provide two key features: First, the same event mechanism is used for both local and remote interactions, i.e. whether the involved components reside within the same address space, or on different machines. This allows for a uniform interaction model that naturally captures the distributed nature of gossip protocols. Second, events can be nested into compound events, to express complex event sequences at different levels of abstraction. (We return to this latter mechanism in Section 4.4.)

Concretely, GOSSIPKIT events take the form of a structured data type (implemented as a plain Java class with appropriate attributes, which is serialised when sent over the network). The key attributes of an event are shown in Table 3. *Event Type* encodes the type of the event, and is the primary means by which events are subscribed to and dispatched to the proper component instance. *Protocol ID* uniquely identifies the protocol instance in which the event was raised. This attribute allows developers to isolate event flows in the case of co-existing protocols. *Protocol ID* also allows for protocol composition, by allowing co-existing protocol instances to interact, as happens for instance when a peer-sampling mechanism is used within a higher-level gossip protocol [11], [58], [8], [33].

Remote events are supported through the *Receiving Node* attribute, which indicates on which remote node an event should be delivered in a point-to-point network. (This attribute simply remains blank for one-hop broadcasts.) All network messages are implemented as *remote events*. When a remote interaction is required, a *remote event* is raised by the Gossip component, and then passed on to the protocol's Network component, which implements the appropriate transport mechanism.

As in earlier event-driven systems [39], [35], GOSSIPKIT events are able to carry data (*Payload*), both from the sender to the handler (as in traditional event systems), but also back from the handler to the receiver (as a method invocation would). This second capability is only available to local events, in which case the sending components is blocked until an answer is received. It is supported by the *Sending Node* and *Event Source* attributes, which identify the component that raised an event, and allow data to be returned to the event's originator. This capability is used for instance when the Gossip component retrieves gossiping data from the State component, or when the State component invokes the State Process component to update its data content.

To fulfill their function, each component role of Figure 1 reacts to a set of prescribed event types in well-defined ways. These prescribed events form the *event interface* of a component role. In total, GOSSIPKIT uses 11 event types to realise the interfaces of Figure 1. The richest event interface is that of the State component role, which responds to 6 types of event (see below), while most other roles (e.g. State Process or Decision) only respond to one event. The Gossip role is a special case. This is because, although it only responds to two events (⋆Gossip[3] and ⋆Forward events), Gossip relies heavily on nested events to propose a rich set of potential behaviours, an approach we describe in more detail below.

4.4 The State and Gossip components

In the following we present the interfaces of two key component roles—that of State and Gossip—to illustrate how events contribute to the genericity and simplicity of our framework.

4.4.1 The State component

To fulfill its role as a node's local data store, a State component responds to ⋆Get, ⋆Add, ⋆Contains, and ⋆Remove events that act on a table made of rows and columns. State also supports ⋆CompareAndRequest and ⋆StateCompression events. ⋆CompareAndRequest is akin to a *diff* operator which is used for pull-based incremental updates as in the Bimodal Gossip protocol [23]. Finally, ⋆StateCompression requests the State component to compress its state (e.g. as in peer-sampling or topology construction algorithms [11], [20], [58], [8]), and delegates the operation to the State Process role.

This set of six events, and the underlying row-and-column data model they support, can accommodate a large variety of protocols, simply by configuring the number of rows and columns and the type of data stored in each cell of the State component's table.

4.4.2 The Gossip component & nested events

The Gossip component orchestrates the execution of gossip rounds by raising events as appropriate for a particular protocol (labels **b**, **c**, **e**, **f** in Fig. 1). The Gossip component further serves as the entry point into a protocol, either via the Periodic Trigger component for periodic protocols, or directly when activated from an external entity. Because of this centrality, each protocol might potentially require its own tailor-made version of the Gossip component, causing fragmentation in GossipKit's code base and reducing opportunities for reuse.

We have found that such a fragmentation can be avoided by *factoring out* some of the Gossip component's behaviour into the events it consumes. This factorisation relies on an optional set of *nested event templates* (*nested events* for short) included in ⋆Gossip or ⋆Forward events. These nested events indicate how Gossip should react to an incoming event, and can be seen as a very basic form of *scripting*, by which simple event flows are factored out of the Gossip component's code, and moved to the description of interactions occurring within the framework. Because nested events can be sent on the network, this also offers a simple case of *code mobility*, through which nodes might influence each other's behaviour to realise distributed interaction patterns.

Note that nested events do raise the issue of forged messages sent by malicious nodes. Such messages could disrupt a protocol by causing the Gossip component to perform unintended interactions. This danger is however inherent to distributed systems and does in fact exists even with plain distributed events. Although we do not discuss it for lack of space, this

3. In the following, we start events names with a star (⋆) to distinguish them from component types.

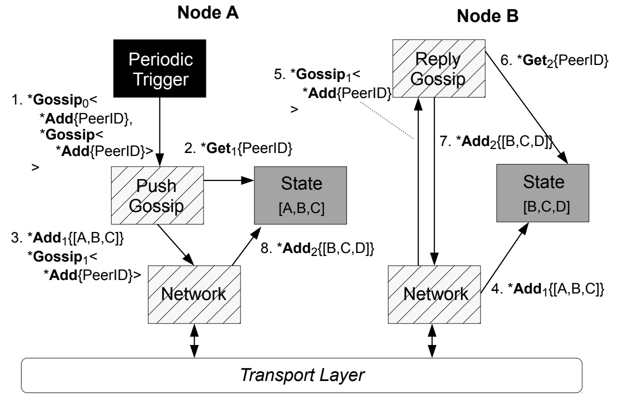


Fig. 3: Using nested events to realise a push-pull interaction

issue would obviously require appropriate protection (using for instance cryptographic signatures) in a production environment.

To illustrate this mechanism in more detail, Figure 3 shows how nested events can be used to perform a *periodic push-pull* exchange between two nodes **A** and **B**. On this figure, event instances are noted with indices (⋆Gossip$_0$), to distinguish them from event types (⋆Gossip). The data carried by events is shown in curly brackets, e.g. ⋆Get$_1${PeerID}, and nested events are noted within angular brackets, e.g. ⋆Gossip$_1$<⋆Add{PeerID}>.

A nested event tells a Gossip component which event should be gossiped on the network. For instance, a Gossip component receiving an event ⋆Gossip$_k$<⋆A,⋆B,⋆C> should instantiate three events: ⋆A$_x$,⋆B$_w$,⋆C$_z$, and propagate them to neighbouring nodes. Nested events are specified as *templates* at this stage, meaning they can refer to fields or columns of a node's state by name. When a nested event is instantiated, those fields and columns names are *expanded* by replacing them with the actual content of the local node's state. For instance, on receiving ⋆Gossip$_1$<⋆Add{PeerID}>, a Gossip component will first retrieve the PeerID column of its local state (e.g. [B,C,D]), and instantiate an ⋆Add event with the actual data: ⋆Add$_2${[B,C,D]} (interactions **5**, **6**, and **7** in Figure 3), before propagating ⋆Add$_2$ on the network.

Using these mechanisms, the *pull-push* interaction between **A** and **B** in Figure 3 starts when **A**'s Gossip component receives a ⋆Gossip$_0$ event with two nested events: ⋆Add{PeerID} and ⋆Gossip<Add{PeerID}> (interaction **1** on Figure 3). On receiving this event, A's Gossip component selects a random peer (**B**, selection not shown) and instantiates both nested events: ⋆Add$_1${[A,B,C]} and ⋆Gossip$_1$ <Add{PeerID}>. ⋆Add$_1$ is expanded in this process by retrieving the PeerID's from **A**'s state (⋆Get$_1${PeerID}, interaction **2**). The two nested events are then triggered on the selected peer **B** as remote events (interaction **3**). The same process repeats itself when node **B** receives ⋆Gossip$_1$<Add{PeerID}> (interactions **5**, **6**, and **7**), except this time **B**'s Gossip component is configured to only send events back to **A** ("Reply" Gossip).

The use of nested components and a simple set of options (for instance to distinguish between a "Push" and "Reply" gossiping behaviour in Figure 3), allows us to provide a single

```
1 <component name="State" id="State_sample">
2   <parameter name="Columns"    type="List"   value="[NodeID, Profile]" />
3   <parameter name="Size"       type="int"    value="5" />
4   <parameter name="PrimaryKey" type="String" value="NodeID" />
5 </component>
```

Fig. 4: An excerpt from the GOSSIPKIT configuration file for RPS [20]

Role	Pre-defined components
Gossip	1
State	1
PeriodicTrigger	1
Network	3
Peer Selection	2
State Process	5

TABLE 4: Predefined components per GOSSIPKIT role

generic implementation of the Gossip component role in GOSSIPKIT. This implementation can then be instantiated multiple times within the same node with different options and nested events to implement a large range of distributed interaction patterns. (For instance, on Figure 3, node **B** would also possess its own "Push Gossip" instance, which is not shown, to trigger pull-push interactions like **A** does.) We provide more examples of this strategy in Section 5 when we evaluate GOSSIPKIT.

4.5 Implementation details and use

We have implemented GOSSIPKIT in Java using OPENCOM, a lightweight and reflective component engine developed at Lancaster [17], [69][4]. OPENCOM components take the form of plain Java objects endowed with specialised Java interfaces to support their dynamic manipulation: creation, binding, unbinding, destruction, and introspection. This manipulation occurs through a component runtime (a singleton object) that provides operations such as createInstance(), deleteInstance(), connect(), and disconnect().

To use GOSSIPKIT, a developer first loads and instantiates a GOSSIPKIT *configuration* into a singleton object called GossipKit (itself an OpenCom component). This configuration describes which components to instantiate, and how they should be bound together. To realise such a configuration, GOSSIPKIT comes with 13 predefined component implementations that realise the roles of Fig. 1. (The breakdown of these components is shown in Table 4.) This set of predefined components can be extended with new components to realise new protocols. This happens by creating a Java class that implements the required GOSSIPKIT and OPENCOM interfaces and adding it to a Java package reserved for this purpose.

Figure 4 shows an excerpt of a GOSSIPKIT configuration (in XML, slightly simplified for readability) for RPS [20]. This excerpt declares a State component instance, which uses the generic state implementation (Sec. 4.4). Lines 2 to 4 describe the type of state to be maintained: Here a list of NodeIDs, each associated with some profile information (line 2).

The design of GOSSIPKIT is not tied explicitly to XML. Other mark-up languages (e.g. JSON, or YAML) could be used for a more compact representation. In addition, and although we do not discuss this aspect in this paper for space reasons, GOSSIPKIT can also be programmed using a tailor-made domain specific language (DSL) called WHISPERS that reifies the underlying family of behaviours captured by the framework [70], [71].

4. http://sourceforge.net/projects/gridkit/files/OpenCOM/

5 EVALUATION

Our evaluation of GOSSIPKIT is both qualitative and quantitative. We first assess the *genericity* of GOSSIPKIT in terms of *configurability* and *reuse* in Section 5.2. We then use software and performance metrics to measure GOSSIPKIT's simplicity (Sec. 5.3), and run-time overheads (Sec. 5.4). We finally demonstrate GOSSIPKIT's reconfigurability, to illustrate one of the direct benefits of using components to implement gossip-based systems (Sec. 5.5).

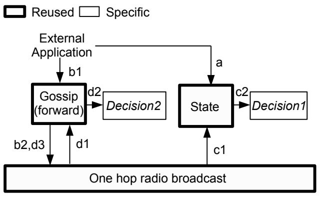
5.1 Evaluation approach

To provide concrete experimental data, we implemented a representative set of *eight* gossip protocols both with GOSSIPKIT, and directly in Java. These eight protocols (Gossip[1&2] [5], SCAMP [28], RPS [20], Anti Entropy (as found in the Bimodal Multicast protocol) [23], Averaging [12], Ordered Slicing [4], and T-Simple) are shown in bold in Table 2. T-Simple is a basic case of *topology construction* which is derived from T-Man [11]: T-Simple works like T-Man except that nodes select the peers with which they communicate uniformly at random (using a peer sampling service), rather than in the current T-Man view as T-Man does. These protocols cover the key patterns introduced in Section 3, and involve each of the six alternative strategies (reactive/periodic, push/pull, one-to-one/one-to-many) that underlie these patterns.

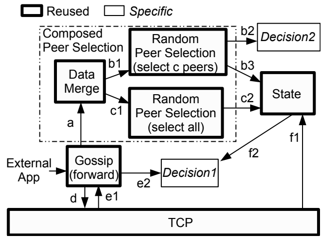
5.2 Configurability and reuse

A well-designed component framework should be *configurable*, and allow developers to realise different instances of the target domain by rearranging the framework's default components, with a minimal amount of specific code. As a collateral bonus, a configurable framework implies that the same code is reused across multiple protocols. This reuse is beneficial because it saves development efforts, and fosters software quality (by exposing the same code to different contexts, and raising the pay-off of each bug correction).

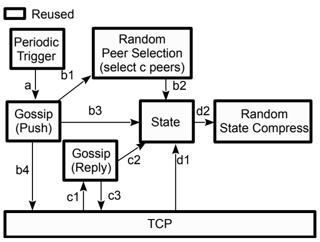
Figure 5 and Table 5 illustrate the configurability and reuse of GOSSIPKIT when applied to the 8 protocols used in this



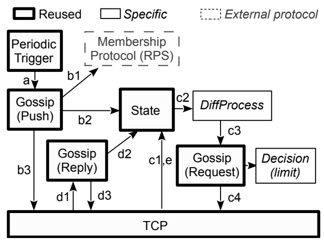
(a) Gossip1,2



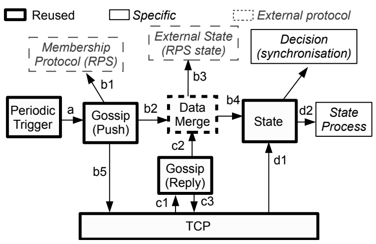
(b) SCAMP



(c) RPS



(d) Anti Entropy (BM)



(e) Averaging, Ordered Slicing, T-Simple (with Data Merge)

Fig. 5: Example configurations of GossipKit

evaluation. Figure 5 shows how GossipKit's architecture is instantiated for each of these protocols, highlighting which components are reused (without any modification), and which must be specifically developed (by sub-classing a default template class). Table 5 tabulates for each protocol the amount of component code shared with at least another protocol against component code unique to this particular protocol.

As can be seen, GossipKit is both highly configurable and reusable, allowing the realisation of a diverse set of protocols by simply rearranging the configuration of existing components. The required amount of specialisation ranges from none for simple protocols (RPS, Fig. 5c), to three components out of eight for more complex examples (Averaging, and Ordered Slicing Fig. 5e). The two component implementations most often reused are those of Gossip and State which we have discussed in Section 4.4. By contrast, all Decision Components are customised, as they often capture design decisions that are unique to the protocol at hand. Components that are specifically developed for one protocol can be saved in the repository for later reuse in other protocols. This allows for a growing pool of components to be built, which can potentially be rearranged to create new protocols with novel capabilities.

Similarly, and although the study only covers 8 protocols, the reuse rate (the proportion of reused component code) ranges from 100% (RPS) to 78.7% (T-Simple), and remains particularly high, with a weighted average of 88.1%. This reuse would likely increase if additional protocols were added, as more commonalities would show up.

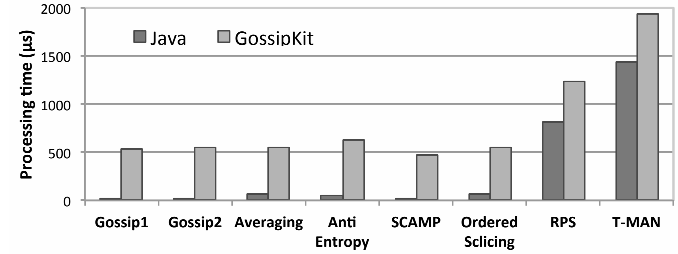


Fig. 6: Local processing time of 8 protocols (μs)

5.3 Simplicity

The configurability and reusability of a framework might come at the cost of a higher complexity, with much effort needed to select, specialise, and integrate components into a working solution [72]. To evaluate GossipKit's effect in this respect, we compared for each protocol the size of its GossipKit configuration (XML) against that of its original monolithic Java implementation (Table 6).

If one assumes, as is reasonable to believe here for XML and Java, that programming efforts are roughly proportional to code size, GossipKit allows for a much more direct construction of protocols than plain Java (by a factor of five).

5.4 Run-time overheads

Compared with a direct implementation in a language like Java, components inevitably add overheads, in terms of execution time and memory usage. This is because the explicit bindings that connect components, and the events used in their interactions incur additional steps in the execution of a GossipKit protocol instance.

5.4.1 *Execution time overhead*

Figure 6 compares the average execution times of GossipKit and plain Java. These times correspond to the duration of one gossip round, measured locally, and do not include any network costs. These times were obtained on a Windows XP SP2 computer with 512 Kbytes of RAM and one 1.73 GHz mono-core processor, using the Java 1.6 SE from Oracle/SUN. Measurements were repeated 50 times and averaged.

These results show that all GossipKit implementations run substantially slower than direct Java versions. We speculate that the difference is mainly caused by the OpenCom runtime, and more specifically by its heavy use of the Java reflective API. However, the overhead incurred (0.5 ms on average) remains much smaller than the typical network latency of wide-area networks (from tens to hundreds of milliseconds), and comparable to that of local-area networks (a fraction of millisecond). These overheads could have an effect on reactive protocols running in a local-area set-up, with stringent execution bounds. In general however, these values remain acceptable, in particular if one considers the low specs of the machine we have used, and the fact that most gossip protocols run at periods of a few seconds to a few minutes, i.e. several orders of magnitude higher that the observed overheads.

Protocol	Reused (LoC)	Specific (LoC)	Reuse Rate	Reused Comp.	Specific Comp.
Gossip1 [5]	626	134	82.3%	3	2
Gossip2 [5]	626	138	81.9%	3	2
SCAMP [28]	888	120	88.1%	6	2
RPS [20]	1221	0	100%	7	0
Anti Entropy (BM) [23]	1349	56	96.0%	7	1
Averaging [12]	1102	152	87.9%	6	2
Ordered Slicing [4]	1102	178	86.1%	6	2
T-Simple (Sec. 5.1)	1144	309	78.7%	6	2
Average	**1007**	**136**	**88.1%**	5.5	1.63

TABLE 5: Reused achieved by GOSSIPKIT

Protocol	GOSSIPKIT (XML LoC)	Java (LoC)	Effort Ratio
Gossip1	39	277	14.1%
Gossip2	39	279	14.0%
SCAMP	88	463	19.0%
RPS	81	439	18.5%
Anti Entropy (BM)	100	544	18.4%
Averaging	85	466	18.2%
Ordered Slicing	85	471	18.0%
T-Simple	93	491	18.9%
Average	**76.3**	**424**	**18.0%**

TABLE 6: Implementation effort vs. Java

5.4.2 *Memory footprint*

One of the aims of GOSSIPKIT is to support gossip protocols across a wide range of networks and devices. This implies that its memory requirements should be reasonable. Table 7 reports the static memory footprint of some of the key parts of GOSSIPKIT (measured as the size of the compiled Java classes). The table distinguishes between the GOSSIPKIT runtime, which provides the execution context for GOSSIPKIT components (mainly the framework proper and its event engine), and key GOSSIPKIT components. Table 8 shows how these numbers translate into the static memory footprint of the eight protocols of this evaluation. On average, each protocol consumes less than 31Kbytes of static memory. This overhead is comparable to similar component-based platforms that target mobile devices (e.g. ReMMoC [73], MANETKit [74]), which typically consume about 30 - 100 Kbytes of memory.

Turning now to dynamic overheads, Table 9 presents the dynamic memory consumed by GOSSIPKIT for our eight gossip protocols, compared with the memory usage of the monolithic versions implemented in Java (measured with JProfiler™ 5).

Overall, GOSSIPKIT consumes on average 35.3% more memory than pure Java. Memory usage remains however under 14,000 Kbytes for all eight protocols. On further analysis, using HPROF [75] and the tool ProfVis [76], this substantial overhead seems to be predominantly caused by the many intermediate meta-data (HashMaps, Lists) created to handle reflective calls in OPENCOM, process events in GOSSIPKIT, and store protocol data. Most of the code uses verbose structures such as String and Integer objects for such information, and could probably be further optimised. This is however comparable to the memory consumption observed in other frameworks based on OPENCOM [47], [74], and acceptable for modern mobile devices, and high-end embedded computers[5].

5.5 Reconfigurability

In addition to reuse and compactness, GOSSIPKIT also brings the traditional advantages associated with component frameworks, such as the ability to reason about configurations, and the mechanisms to reconfigure a running deployment. To illustrate this last point, we present a simple scenario of dynamic reconfiguration with GOSSIPKIT. The scenario involves

Runtime	Static Memory (Bytes)
GOSSIPKIT Framework	8,692
Generic Interfaces	3,216
Event Handler	3,810
Event Handler Registry	5,276
Component	**Static Memory (Bytes)**
Gossip Component	6,861
State Component	6,612
Random Peer Selection Comp.	4,216
TCP Network Component	5,425

TABLE 7: Static memory footprint (compiled bytecode)

three GOSSIPKIT instances and two sequential reconfigurations (Figure 7). The target system is made of 100 nodes deployed in a 10×10 grid, and uses the Jist/SWANS simulator[6] to simulate a fixed network, with gossip rounds set to last 5s, network latencies varying uniformly between 50 and 100ms.

This experiment uses a simple mechanism for distributed reconfigurations that leverages the periodicity of RPS and T-Simple (presented in Sec. 5.1): One node is selected as a reconfiguration driver and generates a *reconfiguration script* which details the set of component-based operations to be performed (loading and unloading components, unbinding and binding events). Reconfiguration scripts are piggybacked on the message of currently running gossip protocols in order to reach all nodes.

Initially all nodes run the RPS protocol [32] to maintain a random graph for peer sampling (Figure 7a). The first reconfiguration consists in launching an implementation of T-Simple to construct a *ring* topology. Because T-Simple relies on RPS to sample peers, the reconfiguration instantiates T-Simple on top of the running RPS. The reconfiguration script is triggered on node 0, propagates through the network, and eventually converges to a ring topology (Figures 7b and 7c). Once a ring topology has been constructed, a second reconfiguration is triggered to use a second implementation of T-Simple to build a *grid* topology (Figures 7d and 7e).

This experiment demonstrates GOSSIPKIT's ability to support reconfigurations at different levels of granularity. The first reconfiguration is coarse-grained: it deploys an entirely new protocol (i.e. T-Simple for constructing a ring topology) atop

5. E.g. a $58 \times 17 \times 4.2$mm DuoVero Gumstix board with 1GB of RAM.

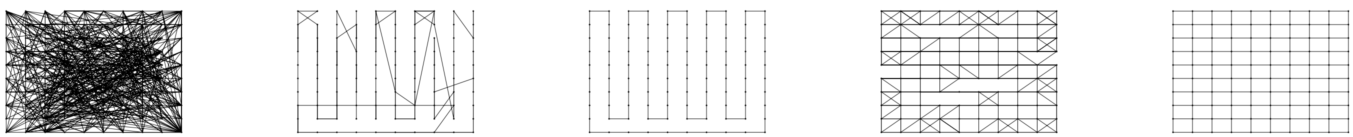
6. http://jist.ece.cornell.edu/

	Protocol	Bytes
Simple	Gossip1	19,885
	Gossip2	20,532
	SCAMP	25,923
	RPS	29,941
Composite	Anti Entropy (BM)	34,162
	Averaging	36,209
	Ordered Slicing	36,398
	T-Simple	41,529
	Average	**30,572**

TABLE 8: Byte code size of the eight gossip protocols. The byte code size of the four composite protocols includes the size of RPS.

	Protocol	Java (Kbytes)	GOSSIPKIT (Kbytes)	Overhead
Simple	Gossip1	8,832	12,968	46.8%
	Gossip2	8,864	12,976	46.3%
	SCAMP	10,012	13,308	32.9%
	RPS	10,016	13,316	32.9%
Composite	Anti Entropy (BM)	10,120	13,376	33.5%
	Averaging	10,136	13,380	32.0%
	Ordered Slicing	10,128	13,486	33.2%
	T-Simple	10,540	13,572	28.8%
	Average	**9,831**	**13,298**	**35.3%**

TABLE 9: Dynamic memory usage of the eight gossip protocols. The measurements of the two gossip protocols that run on wireless ad hoc networks, Gossip1 and Gossip2, do not include the Jist/SWANS simulator.



(a) RPS alone

(b) 5^{th} round: RPS + T-Simple[ring]

(c) 11^{th} round: RPS + T-Simple[ring]

(d) 20^{th} round: RPS + T-Simple[grid]

(e) 23^{th} round: RPS + T-Simple[grid]

Fig. 7: Reconfiguration: dynamic deployment of T-Simple[ring], followed by a reconfiguration into T-Simple[grid]

Reconfiguration Type	CPU Overhead (μs)
Single component (Figures 7d-7e)	2,036
Entire protocol (Figures 7a-7c)	13,811

TABLE 10: The time used for reconfiguration

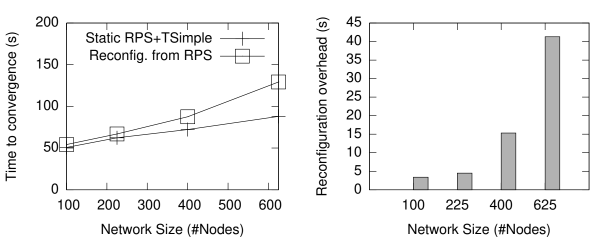


Fig. 8: End-to-end reconfiguration overhead (T-Simple[ring])

RPS, instantiating 8 new components and 10 new bindings. The second one is fine-grained, and only involves the State Process component of T-Simple and two bindings.

The local reconfiguration times (without networking costs) for the two reconfigurations in the above scenario are shown in Table 10. The end-to-end overhead of the first reconfiguration (the dynamic deployment of T-Simple[ring]) compared to a static deployment of RPS and T-Simple[ring] is shown on Fig. 8 for various network sizes. All measures are averaged over 50 runs. These numbers demonstrate the ability of GOSSIPKIT to support a substantial reconfiguration (here the dynamic deployment of T-Simple) under low local overhead (less than 14 ms). The end-to-end overheads of Fig. 8 further show the small incidence of the local reconfiguration time (14ms) on the overall system performance, which is mainly driven by the duration of individual rounds (5s).

6 CONCLUSION

In this paper we have presented GOSSIPKIT, a modular and generic component framework for the realisation of gossip-based systems. GOSSIPKIT's architecture is grounded in a principled survey of a large set of existing gossip protocols, covering both fixed and wireless networks. This survey has led us to propose a set of three design dimensions, and four recurring design patterns underlying most gossip-based protocols. GOSSIPKIT embodies those dimensions and patterns in a concrete reusable architecture that is both simple and generic.

GOSSIPKIT lies in the direct continuation of the many works conducted at Lancaster on fine-grained structures—that is component-based architectures—in a variety of distributed systems areas: protocol stacks, router software, overlays. GOSSIPKIT demonstrates that a fined-grained structural decomposition is also applicable to distributed probabilistic systems, and opens up a number of interesting questions regarding the adaptation and composition of gossip-based systems.

For instance, component architectures—coarse grained *and* fine grained—have been acknowledged to support cross-layer optimisation well. However, whereas this is an area that is well recognised in the literature, there are fewer examples of real exploitation. Some of the structures we have proposed for GOSSIPKIT would seem particularly promising in this area, for example to automatically reason about synergies and conflicts between gossip-based systems coexisting within the same infrastructure [33]. Moving beyond gossip protocols, we have also started to work on emergent middleware and dynamic interoperability [77], and it would be interesting to translate this work to the dynamic interoperability of gossip protocols. Finally, we see a broader opportunity to investigate how opportunistic mechanisms such as gossip can be integrated into larger and more complex distributed systems—thus feeding into an understanding of systems-of-systems.

ACKNOWLEDGMENTS

This work has been supported by the ESF MiNEMA program (Ref. 1954) and partially by the EU FP7 ReSIST Network of Excellence (n. 026764).

REFERENCES

[1] S. Lin, F. Taïani, and G. S. Blair, "Facilitating gossip programming with the gossipkit framework," in *Proc. of the 8th IFIP Int. Conf. on Dist. App. and Interoperable Sys. (DAIS'08)*, 2008, pp. 238–252.

[2] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proc. of the 6th Annual ACM Symp. on Principles of Dist. Comp.* ACM, August 1987, pp. 1–12.

[3] J. Holliday, R. Steinke, D. Agrawal, and A. E. Abbadi, "Epidemic algorithms for replicated databases," *IEEE Trans. on Knowledge and Data Eng.*, vol. 15, no. 3, pp. 1218–1238, May/June 2003.

[4] M. Jelasity and A.-M. Kermarrec, "Ordered slicing of very large-scale overlay networks," in *Proc. of the 6th IEEE Int. Conf. on Peer-to-Peer Computing*. IEEE, September 2006, pp. 117–124.

[5] Z. J. Haas, J. Y. Halpern, and L. Li, "Gossip-based ad hoc routing," *IEEE/ACM Trans. Netw.*, vol. 14, pp. 479–491, June 2006.

[6] X. Hou, D. Tipper, and S. Wu, "A traffic-aware power management protocol for wireless ad hoc networks," *Journal of Communication*, vol. 1, no. 2, pp. 38–48, 2005.

[7] ——, "A gossip-based energy conservation protocol for wireless ad hoc and sensor networks," *Network & Sys. Mngmt.*, vol. 14, pp. 381–414, 2006.

[8] S. Voulgaris and M. v. Steen, "Epidemic-style management of semantic overlays for content-based searching," in *Proc. of the 11th Int. Euro-Par Conf. on Parallel Proc. (Euro-Par'05)*, 2005, pp. 1143–1152.

[9] M. Bertier, D. Frey, R. Guerraoui, A.-M. Kermarrec, and V. Leroy, "The gossple anonymous social network," in *Proc. of the ACM/IFIP/USENIX 11th Int. Conf. on Middleware (Middleware'10)*, 2010, pp. 191–211.

[10] A.-M. Kermarrec and M. van Steen, "Gossiping in distributed systems," *ACM SIGOPS Operating System Review*, vol. 41, no. 5, pp. 2–7, 2007.

[11] M. Jelasity, A. Montresor, and O. Babaoglu, "T-man: Gossip-based fast overlay topology construction," *Comput. Netw.*, vol. 53, no. 13, pp. 2321–2339, Aug. 2009.

[12] ——, "Gossip-based aggregation in large dynamic networks," *ACM TOCS*, vol. 23, no. 3, pp. 219–252, 2005.

[13] Y. Sasson, D. Cavin, and A. Schiper, "Probabilistic broadcast for flooding in wireless mobile ad hoc networks," in *Proc. of IEEE Wireless Comm. and Netw. Conf. (WCNC'03)*, vol. 2, 2003, pp. 1124–1130.

[14] J. Luo, P. Eugster, and J.-P. Hubaux, "Route driven gossip: Probabilistic reliable multicast in ad hoc networks," in *Proc. of the 22nd Annual Joint Conf. of the IEEE Comp. and Comm. Soc. (INFOCOM'03)*, vol. 3, March 2003, pp. 2229–2239.

[15] J. Bosch, C. Szyperski, and W. Weck, "Summary of wcop'98," in *Proc. of the 3^{rd} Int. Workshop of Component-Oriented Programming (WCOP'98) (co-located with ECOOP)*, ser. TUCS general publication, no. 10, 1998, pp. 1–5.

[16] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, 2002.

[17] P. Grace, G. Coulson, G. Blair, L. Mathy, W. K. Yeung, W. Cai, D. Duce, and C. Cooper, "Gridkit: Pluggable overlay networks for grid computing," in *Proc. of Int. Symp. on Dis. Objects and App. (DOA)*. Springer, 2004, pp. 1463–1481.

[18] P. K. McKinley, U. I. Padmanabhan, and N. Ancha, "Experiments in composing proxy audio services for mobile users," in *Proc. of the IFIP/ACM Int. Conf. on Dis. Sys. Platforms (Middleware 2001)*, November 2001, pp. 99–120.

[19] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A component-based middleware platform for reconfigurable service-oriented architectures," *Software: Practice and Experience*, pp. n/a–n/a, 2011.

[20] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM TOCS*, vol. 25, August 2007.

[21] P. T. Eugster and R. Guerraoui, "Probabilistic multicast," in *Proc. of the 2002 Int. Conf. on Dependable Sys. and Networks (DSN'02)*. IEEE, 2002, pp. 313–324.

[22] D. Kempe, J. Kleinberg, and A. Demers, "Spatial gossip and resource location protocols," in *Proc. of the 33rd Annual ACM Symp. on Theory of Computing (STOC'01)*. ACM, 2001, pp. 163–172.

[23] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM TOCS*, vol. 17, pp. 41–88, May 1999.

[24] R. V. Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Proc. of the 1st IFIP Int. Conf. on Dis. Sys. Platforms and Open Dis. Proc. (Middleware'98)*. Springer, 1998, pp. 55–70.

[25] Étienne Rivière, R. Baldoni, H. Li, and J. Pereira, "Compositional gossip: A conceptual architecture for designing gossip-based applications," *ACM SIGOPS Operating System Review*, vol. 41, no. 5, pp. 43–50, 2007.

[26] K. Birman, "The promise, and limitations, of gossip protocols," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 5, pp. 8–13, Oct. 2007.

[27] I. Gupta, A.-M. Kermarrec, and A. J. Ganesh, "Efficient and adaptive epidemic-style protocols for reliable and scalable multicast," *IEEE Trans. on Parallel Dis. Sys.*, vol. 17, no. 7, pp. 593–605, 2006.

[28] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, "Peer-to-peer membership management for gossip-based protocols," *IEEE Trans. Comput.*, vol. 52, pp. 139–149, February 2003.

[29] ——, "Scamp: Peer-to-peer lightweight membership service for large-scale group communication," in *Proc. of the 3rd Int. Workshop on Networked Group Commu.*, 2001, pp. 44–55.

[30] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec, "Lightweight probabilistic broadcast," *ACM TOCS*, vol. 21, pp. 341–374, November 2003.

[31] I. Gupta, R. V. Renesse, and K. P. Birman, "Scalable fault-tolerant aggregation in large process groups," in *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'01)*, June 2001, pp. 433–442.

[32] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "The peer sampling service: experimental evaluation of unstructured gossip-based implementations," in *Proc. of the 5th ACM/IFIP/USENIX Inf. Conf. on Middleware (Middleware'04)*. Springer, 2004, pp. 79–98.

[33] S. Lin, F. Taïani, and G. S. Blair, "Exploiting synergies between coexisting overlays," in *Proc. of the 9th IFIP Int. Conf. on Dis. App. and Interoperable Sys. (DAIS'09)*, June 2009, pp. 1–15.

[34] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr, "Building adaptive systems using ensemble," *Softw. Prac. and Exp.*, vol. 28, no. 9, pp. 963–979, 1998.

[35] M. A. Hiltunen and R. D. Schlichting, "The cactus approach to building configurable middleware," in *Workshop on Dependable Sys. Middleware & Group Comm. (DSMGC'00)*. IEEE CS Press, October 2000.

[36] H. Miranda, A. Pinto, and L. Rodrigues, "Appia, a flexible protocol kernel supporting multiple coordinated channels," in *in Proc. 21st Int. Conf. on Dis. Comp. Sys. (ICDCS-21)*. IEEE, 2001, pp. 707–710.

[37] K. Birman and R. Cooper, "The ISIS project: Real experience with a fault tolerant programming system," *SIGOPS Operating Sys. Review*, vol. 25, no. 2, pp. 103–107, 1991.

[38] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr, "A framework for protocol composition in horus," in *14th Annual ACM Symp. on Princ. of Dist. Comp. (PODC'95)*. ACM, 1995, pp. 80–89.

[39] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu, "Coyote: A system for constructing fine-grain configurable communication services," *ACM TOCS*, vol. 16, no. 4, pp. 321–366, 1998.

[40] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM TOCS*, vol. 26, no. 1, pp. 1:1–1:42, Mar. 2008.

[41] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Software Practical and Experience*, vol. 36, pp. 1257–1284, September 2006.

[42] P. Eugster, P. Felber, and F. L. Fessant, "The "art" of programming gossip-based systems," *ACM SIGOPS Operating System Review*, vol. 41, no. 5, pp. 37–42, 2007.

[43] L. Princehouse and K. Birman, "Code-partitioning gossip," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 40–44, Jan. 2010.

[44] D. Gay, P. Levis, R. R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," *SIGPLAN Notices*, vol. 38, no. 5, pp. 1–11, 2003.

[45] D. Hughes, K. Thoelen, W. Horré, N. Matthys, J. D. Cid, S. Michiels, C. Huygens, and W. Joosen, "LooCI: a loosely-coupled component infrastructure for networked embedded systems," in *Proc. of the 7th Int. Conf. on Advances in Mobile Computing and Multimedia (MoMM'09)*. ACM, 2009, pp. 195–203.

[46] B. Porter and G. Coulson, "Lorien: a pure dynamic component-based operating system for wireless sensor networks," in *Proc. of the 4th Int. Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens'09)*. ACM, 2009, pp. 7–12.

[47] P. Grace, D. Hughes, B. Porter, G. S. Blair, G. Coulson, and F. Taiani, "Experiences with open overlays: A middleware approach to network heterogeneity," in *Proc. of the 3rd ACM SIGOPS/EuroSys European Conf. on Computer Systems 2008 (Eurosys 2008)*, 2008, pp. 123–136.

[48] J.-C. Fabre, M.-O. Killijian, and T. Pareaud, "Towards on-line adaptation of fault tolerance mechanisms," in *Eighth European Dependable Computing Conference (EDCC-8)*, 2010, pp. 45–54.

[49] M. Lin and K. Marzullo, "Directional gossip: Gossip in a wide area network," in *Proc. of European Dependable Computing Conf.* Springer, September 1999, pp. 364–379.

[50] J. Leitao, J. Pereira, and L. Rodrigues, "Epidemic broadcast trees," in *Proc. of the 26th IEEE Int. Symp. on Reliable Dis. Sys. (SRDS'07)*. IEEE, 2007, pp. 301–310.

[51] N. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues, "Emergent structure in unstructured epidemic multicast," in *Proc. of the 37th IEEE/IFIP Int. Conf. on Dependable Sys. and Networks*, vol. 0. Los Alamitos, CA, USA: IEEE, 2007, pp. 481–490.

[52] A. Nedos, K. Singh, R. Cunningham, and S. Clarke, "A gossip protocol to support service discovery with heterogeneous ontologies in manets," in *Proc. of the Third IEEE Int. Conf. on Wireless and Mobile Comp., Netw. and Comm. (WIMOB'07)*. IEEE, 2007, p. 53.

[53] K. Guo, M. Hayden, R. van Renesse, W. Vogels, and K. P. Birman, "Gsgc: An efficient gossip style garbage collection scheme for scalable reliable multicast," Cornell University, Ithaca NY, 14853, Technical Report TR97-1656, December 1997.

[54] J. Pereira, L. Rodrigues, M. J. Monteiro, R. Oliveira, and A.-M. Kermarrec, "Neem: Network-friendly epidemic multicast," in *Proc. of the 22th IEEE Symp. on Reliable Dist. Sys.*, October 2003, pp. 15–24.

[55] P. Santos and J. O. Pereira, "Neem: Network-friendly epidemic multicast," https://github.com/jopereira/neem, accessed 2/7/2013.

[56] K. Jenkins, K. Hopkinson, and K. Birman, "A gossip protocol for subgroup multicast," in *Proc. of the 21st Int. Conf. on Dist. Comp. Systems Workshop (ICDCSW'01)*. IEEE, 2001, pp. 25–30.

[57] R. Chandra, V. Ramasubramanian, and K. P. Birman, "Anonymous gossip: Improving multicast reliability in mobile ad-hoc networks," in *Proc. of the 21st Int. Conf. on Dist. Comp. Sys. (ICDCS'01)*. IEEE, April 2001, pp. 275–283.

[58] S. Voulgaris, D. Gavidia, and M. van Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005.

[59] J. Leitao, J. Pereira, and L. Rodrigues, "Hyparview: A membership protocol for reliable gossip-based broadcast," in *Proc. of the 37th Annual IEEE/IFIP Int. Conf. on Dep. Sys. and Networks (DSN'07)*. IEEE, 2007, pp. 419–429.

[60] A. Montresor, M. Jelasity, and O. Babaoglu, "Chord on demand," in *Proc. of the IEEE Int. Conf. on Peer-to-Peer Comp (P2P'05)*. IEEE, 2005, pp. 87–94.

[61] M. Jelasity, W. Kowalczyk, and M. van Steen, "Large-scale newscast computing on the internet," Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, Technical Report IR-CS-006, November 2003.

[62] R. Melamed and I. Keidar, "Araneola: A scalable reliable multicast system for dynamic environments," *J. Parallel Distrib. Comput.*, vol. 68, no. 12, pp. 1539–1560, Dec. 2008.

[63] J. Liu and M. Zhou, "Tree-assisted gossiping for overlay video distribution," *J. of Multimedia Tools & App.*, vol. 29, no. 3, pp. 211–232, 2006.

[64] E. Simonton, B. K. Choi, and S. Seidel, "Using gossip for dynamic resource discovery," in *Proc. of Int. Conf. on Parallel Processing (ICPP'06)*. IEEE, August 2006, pp. 319–328.

[65] P. Kyasanur, R. Choudhury, and I. Gupta, "Smart gossip: An adaptive gossip-based broadcasting service for sensor networks," in *Proc. of IEEE Int. Conf. on Mobile Adhoc and Sensor Systems*, vol. 0. Los Alamitos, CA, USA: IEEE, October 2006, pp. 91–100.

[66] R. Beraldi, "The polarized gossip protocol for path discovery in manets," *Journal of Ad Hoc Network*, vol. 6, no. 1, pp. 79–91, 2008.

[67] D. Frey, R. Guerraoui, A.-M. Kermarrec, B. Koldehofe, M. Mogensen, M. Monod, and V. Quéma, "Heterogeneous gossiping," in *Proc. of the Int. Middleware Conf. (Middleware'09)*, November 2009.

[68] X. Hou and D. Tipper, "Gossip-based sleep protocol (gsp) for energy efficient routing in wireless ad hoc networks," in *The 2004 IEEE Wireless Comm. and Netw. Conf. (WCNC 2004)*, vol. 3, 2004, pp. 1305–1310.

[69] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas, "An efficient component model for the construction of adaptive middleware," in *Proc. of the IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware'01)*. Springer, 2001, pp. 160–178.

[70] S. Lin, "Transparent componentisation: A hybrid approach to support the development of contemporary distributed systems," Ph.D. dissertation, Lancaster University, UK, Sep. 2010.

[71] S. Lin, F. Taïani, M. Bertier, G. Blair, and A.-M. Kermarrec, "Transparent componentisation: high-level (re)configurable programming for evolving distributed systems," in *Proc. of the 2011 ACM Symp. on Applied Comp. (SAC'11)*. ACM, 2011, pp. 203–208.

[72] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, Jun. 1992.

[73] P. Grace, G. S. Blair, and S. Samuel, "Remmoc: A reflective middleware to support mobile client interoperability," in *Proc. of Int. Symp. of Distributed Objects and Applications*, October 2003, pp. 1170–1187.

[74] R. Ramdhany, P. Grace, G. Coulson, and D. Hutchison, "Manetkit: Supporting the dynamic deployment and reconfiguration of ad-hoc routing protocols," in *IFIP/ACM/USENIX 10th Int. Middleware Conf.*, 2009.

[75] Oracle, "Hprof: A heap/cpu profiling tool," http://docs.oracle.com/-javase/7/docs/technotes/samples/hprof.html, accessed 24 June 2013.

[76] S. Lin, F. Taïani, T. C. Ormerod, and L. J. Ball, "Towards anomaly comprehension: using structural compression to navigate profiling call-trees," in *Proc. of the 5th Int. Symp. on Soft. visualization (SOFTVIS'10)*. ACM, 2010, pp. 103–112.

[77] G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci, "The role of ontologies in emergent middleware: supporting interoperability in complex distributed systems," in *Proc. of the 12th ACM/IFIP/USENIX Int. Conf. on Middleware (Middleware'11)*. Springer, 2011, pp. 410–430.

François Taïani is a Professor at Université de Rennes 1. He graduated from École Centrale Paris and Universität Stuttgart in 1998, and received his PhD in Computer Science from Université de Toulouse III in 2004. Following an intervening spell at AT&T Labs (USA) in 2004, François worked from 2005 to 2012 at Lancaster University (UK). His research interests include open programming frameworks for extra-large scale and decentralized distributed systems.

Shen Lin graduated from the University of Edinburgh in 2006, and received his PhD in Computer Science from Lancaster University in 2011. He is currently with the Imagineering team at SAP Labs (Shanghai, PRC). His current research interests include cloud computing, and data visualisation.

Gordon Blair is a Professor of Distributed Systems at Lancaster and is also an Adjunct Professor at the University of Tromsøin Norway. His research focuses on reflective and adaptive middleware, model-driven engineering of middleware families and also middleware for highly heterogeneous environments. He has published over 250 papers in his field and is Chair of the Steering Committee of the ACM/ IFIP/ Usenix Middleware conference.

APPENDIX

DESCRIPTION OF THE PROTOCOLS ANALYSED

The following list briefly describes the protocols analysed in Sec. 3 of the paper. For space reasons, some important aspects of the works presented might be left out. These details can be found in the relevant publications.

Anonymous Gossip [57] provides a repair mechanism for unreliable multicast protocols designed for MANETs, such as MAODV. Nodes advertise the messages they have missed (pull interaction), and receive copies buffered by other nodes. One key contribution of the protocol consists in allowing nodes to advertise digests of the messages they have missed without knowing which nodes these digests will reach (hence "anonymous" gossip), a desirable property in MANETs in which group membership is costly to maintain.

Anti Entropy [2] was originally proposed to propagate database updates in large systems. It works in two phases: First, an efficient unreliable broadcast (e.g. UDP broadcast) propagates updates to as many nodes as possible. Then, running in the background, a periodic gossip algorithm repairs the nodes that have missed the original broadcast. To do this, nodes send the content of their database to another randomly chosen node, and recover lost updates using typically a pull or push-pull approach. The original anti-entropy protocol has subsequently led to several variants (e.g. in Bimodal Multicast [23]—the one we implement in our evaluation).

Araneola [62] uses Pattern P2 ("Continuous Dissemination") to construct a balanced random overlay in which all nodes maintain almost the same target degree (k or $k+1$). Araneola combines a periodic trigger with a local condition on the state of nodes: nodes only gossip when their degree diverges from the target value. This can be analysed as a form a guarded periodic gossip.

Averaging [12] uses periodic pairwise (pull-push) exchanges to average a value held by each node (e.g. a sensor reading). In each round, a node n exchanges its current value v_n with a randomly selected peer i. n and i then update their value to be the average of v_n and v_i. As the protocol progresses, the values stored on individual nodes gradually converge to a global average.

Bimodal Multicast [23] (also known as *pbcast*) uses an optimised anti entropy protocol (inspired from [2]) to repair potential message losses caused by an unreliable multicast service (possibly itself implemented as a gossip protocol). To detect losses, nodes periodically disseminate digests of the messages they have received so far. When a node detects it has missed a message, it requests the missing message from the digest's originator. The protocol further includes a number of optimisations (e.g. dropping late requests, limiting re-transmissions occurring in one single round) to improve its robustness.

Cyclon [58] provides a peer-sampling service by periodically shuffling the neighbourhood lists of individual nodes. Each node p maintains a list of c neighbours (known as a *cache*), and swaps during each round a randomly selected sub-list of $\ell < c$ neighbours with a "well-chosen" neighbour q. To deliver good connectivity and freshness guarantees, q is selected to be the oldest neighbour entry (in rounds) in p's cache.

Directional Gossip [49] is a gossip-based multicast protocol that takes into account the topology of a wide-area network, and selects with a higher probability remote peers (e.g. nodes in different local networks) to accelerate the distribution of information.

G-FDS [24] is a gossip-based failure detection protocol based on heartbeats. Each node maintains a list of known other nodes, along with a heartbeat counter, and the last time this counter was increased for each known nodes. Periodically a node p increments its own heartbeat counter, and sends its list to a node q randomly chosen from its view. q merges p's list into its own by keeping the maximum heartbeat for each node. A node whose heartbeat has not increased for more than a threshold T_{fail} period is considered failed. The protocol also contains an optimisation to take into account the underlying topology of the network, by weighting the selection of nodes according to the subnet they belong to.

G-SDP [52] provides a service discovery service for MANETs based on heterogeneous ontologies. Each node periodically gossips the ontology concepts it knows of to a set of random neighbours. A node receiving a new set of concepts matches these concepts to its local ontology, and stores the new concepts. Concepts keep propagating until they reach a time-to-live value (expressed as a maximum number of hops), at which point they are deleted from a node's local view. The dissemination algorithm uses a peer-sampling service similar to RPS.

GSGC [53] (Gossip Style Garbage Collection) is a distributed garbage collection protocol for reliable multicast algorithms. Most reliable multicast algorithms require node to keep copies of messages, which must be garbage-collected once a message has been received by all nodes in the system. GSGC provides a gossip-based solution to this problem that runs in two phases: In a first phase, each node p disseminates a vector R_p of the highest message id $R_p[j]$ it has received from node j, so that p has also received all messages from j before $m_{R_p[j]}$. As the vector R_p propagates, it gets merged with that of other nodes using a minimum operator, and keeping track of which nodes have contributed to it. Once a node detects the vector is complete (all nodes have contributed), the second phase is launched, and the resulting R_{stable} vector disseminated to all nodes.

GSP [6] (Gossip-based Sleep Protocol) is a broadcast protocol for wireless sensor networks (WSN), in which each node decides to enter its sleep mode for a fixed length of time with probability p, or to stay awake for a random time interval. Only nodes that are awake re-broadcast incoming messages, insuring the propagation of messages in the network while saving energy.

T-GSP [7] extends GSP by requiring nodes to stay awake when they are carrying frequent network traffic, thus reducing the probability of breaking active communications. In [7], the authors show that gossip-based sleep protocols can conserve battery power on WSN nodes 25% longer while maintaining the same delivery rate and transmission latency compared to non-gossip solutions such as DSR.

Gossip$_{1,2,3}$ [5] is a family of three broadcast protocols for routing requests in wireless networks (WSNs, MANETs), that use increasingly elaborate decision schemes to decide whether to re-transmit a route request. In its basic version, Gossip$_1$, all nodes re-transmit a received request with a base probability p_1, except during the first k hops of the request, when they re-transmit with a probability of 1. Gossip$_2$ extends Gossip$_1$ by using a boosted probability $p_2 > p_1$ after the first k hops if a node has less than n neighbours. Finally, Gossip$_3$ extends Gossip$_1$ by using a counting mechanism for nodes that originally decides *not* to re-broadcast a request: If these nodes hear less than m re-transmissions of the original message within some time-out period, they re-broadcast the request.

Gravitational Gossip [56] is a gossip-based multicast protocol that uses non-uniform gossiping probabilities. In each round, node n_i has a probability $I_i \times S_j$ to send a message to node n_j, where I_i is the *infectivity* of n_i and S_j the *susceptibility* of n_j. Nodes are organised in "strata", or sub-sets of nodes that have the same *infectivity* and *susceptibility*, so that nodes in stratum (or rating) $r \in [0, 1]$ have a probability r (or almost r) of receiving updates before these updates time out. The *infectivity* and *susceptibility* of each stratum is chosen using an analytic mathematical model of the infection mechanism of updates.

Hierarchical Gossip [27] is a multicast protocol that preferably selects nodes close in the network topology to reduce network load across domain boundaries. To this aim, the protocol uses a leaf-box hierarchy (a tree-like structure constructed using a hash function on nodes) that maps individual network domains to continuous leaf-boxes. Nodes then gossip with decreasing probabilities to the levels of the hierarchy, which correspond to nodes that are potentially increasingly further in terms of network domain.

HyParView [59] provides a peer-membership protocol for application-level multicast services based on flooding. HyParView addresses the problem of periodic peer-membership protocols such as Cyclon [58], or SCAMP [28] that have long update cycles, and cannot react fast enough in case of large-scale node failures. In HyParView each node maintains two partial views of the system: a small-scale *active symmetric* view, that is managed reactively (i.e. is updated immediately when nodes leave or join), and a larger-scale *passive* and *asymmetric* view, that is maintained by periodic random-walk shuffles. When a node in the active view is detected as failed (using TCP's in-built failure detection mechanism), it is replaced by a node from the passive view, thus providing a fast reaction to failures.

NEEM [54], [55] (*Network Friendly Epidemic Multicast*) aims to provide semantic-based congestion control in gossip-based multicast protocols. It enriches a gossip-based multicast protocol with a specialised buffer management technique applicable to connection-oriented point-to-point transport protocols such as TCP. NEEM exploits the congestion control of TCP, while discarding those messages that are the least critical for the application. Discarded messages are advertised to other nodes to stop their propagation. NEEM also implements a lazy dissemination mechanism to recover lost messages [55].

Newscast [61] provides both a membership and information dissemination protocol using a periodic gossip mechanism. Each node maintains a cache of time-stamped news items and information about other peers, and periodically exchanges this whole list with a randomly selected peer, keeping only the c most recent entries of the resulting merged list.

K-Walker [64] is a gossip-based resource discovery. A node requesting a resource randomly disseminates a request to other nodes. The request is propagated until the request expires or an appropriate node is found. This pull-based request dissemination mechanism is enriched with information regarding the resources available at the visited nodes, that is piggy-backed on the requests, and cached by receiving nodes. This cached information is in turn used to bias the dissemination of subsequent requests towards nodes likely to be able to fulfill them.

lpcast [30] (*lightweight probabilistic broadcast*) proposes a reliable broadcast protocol that uses digests to recover missing messages. One of lpcast's key contributions is a smart garbage collection technique that tends to keep message copies in node buffers based on their usefulness for future rounds, rather than delete them randomly. lpcast uses two heuristics to purge message buffers: *age-based purging* is used for event notifications, while *frequency based purging* is used for node subscriptions.

Ordered Slicing [4] provides a partitioning protocol in which the resulting groups are ordered according to a particular measurable property (e.g. bandwidth, workload) of the nodes. Ordered Slicing uses a swap function on pairs of attribute values to order nodes. These pairs are made of two numbers: one of is the property of interest, with the other one is a uniformly distributed random number. The protocol converges to a situation where the order of the random numbers reflects the order of the property of interest, while eliminating any skew that might exist in the distribution of this property. These random numbers can then be used to partition the system in slices proportional to the system's overall size.

PlumTree [50] (*Push-Lazy-Push Multicast Tree*) constructs a broadcast tree within a gossip overlay. Messages are primarily propagated on the broadcast tree using a punctual dissemination pattern (*eager push*). When a failure occurs, however, a lazy dissemination approach is used to recover lost messages and reconstruct the broadcast tree.

Polarized Gossip [66] uses gossip to discover routing path in MANETs. Polarized Gossip uses varying probabilities when re-transmitting messages that depends on the geographical distance from the current node to the message's destination and the distance from the previous hop to the destination. These distances are in turned estimated using periodic one-hop beacons, and a very simple model of the nodes' likely mobility behaviour.

Probabilistic Multicast [21] is a gossip-based multicast protocols that limits the propagation of events to nodes potentially interested in these events, rather than to all nodes in the system. To this aim, nodes are organised in a set of spanning trees using address prefixes, with Boolean predicates embedded into the trees to capture collective node subscriptions. These predicates condition the propagation of events to sub-trees that contain interested nodes using a continuous forward dissemination (periodic push), while the tree is maintained using a continuous polling mechanism (periodic pull).

RDG [14] (*Route Driven Gossip*) extends the routing primitives provided by an on-demand ad-hoc routing protocol (e.g. DSR) to provide a gossip-based reliable multicast service in wireless networks (WSNs, MANETs). The gossip protocol uses point-to-point links constructed by the reactive routing protocol to disseminate information. The protocol predominantly uses the periodic dissemination pattern, extended with a pull mechanism for packets detected as missing.

RPS [20] (*Random Push-pull Blind Peer Selection*) is one of the configurations of the family of peer sampling protocols proposed in [20]. In this configuration, nodes periodically exchange their partial view with a random neighbour, and keep a random fixed-size subset of the merged views.

SCAMP [28] is a peer-membership protocol that uses punctual dissemination to balance the network connectivity so that each node maintains a view of $\log(N)$ evenly distributed random nodes. On receipt of a join request from node i, a SCAMP node n forwards the request to all the nodes in its view. On receipt of a forwarded request, a node j adds node i to its view with probability p, and otherwise forwards i's join request to a random node in its view. This propagation mechanism helps in turn balance the random graph every time a node joins the network.

Smart Gossip [65] provides a wireless broadcast service based on Gossip. Rather than using a single set of parameters for the entire network (as the $\text{Gossip}_{1,2,3}$ family of protocols does [5]), Smart Gossip adapts the probability of gossiping of each node based on the *importance* of this node for the whole network. To this aim Smart Gossip nodes progressively learns the local topological properties of the network by overhearing ongoing broadcasts, and deducing local propagation paths between nodes. Nodes that are identified as hubs on these paths end up broadcasting with a higher probability after an adaptive learning phase.

Spatial Gossip [22] is a broadcast protocol for a (potentially infinite) set of nodes with positions in $\mathbb{R}^D$, so that the probability that a node x contacts a node y decreases polynomially in the distance between x and y. This type of protocol can be analysed formally and shown to exhibit (probabilistic) propagation times in the logarithm of the distance between the source of a message and its receivers.

T-Chord [60] uses T-Man to bootstrap the overlay needed to run the Chord DHT (Distributed Hash Table). Chord combines a ring overlay, with a set of finger links that create the logarithmic routing structure exploited by the DHT. T-Chord constructs the Chord ring using an appropriate distance function, and exploits the list of nodes visited by T-Man to approximate the finger links needed by Chord.

T-Man [11] is a gossip-based topology construction protocol. T-Man assumes that each node has a position in a metric space $\mathcal{E}$, and construct an overlay so that each node n becomes connected to (at least) the k nodes closest to n in $\mathcal{E}$. To achieve this result, each node periodically exchange its top m neighbours with a neighbour chosen among its ψ closest neighbours.

TAG [63] (*Tree-Assisted Gossiping*) combines a standard tree overlay with a bidirectional gossip overlay to disseminate a video stream over numerous nodes. The gossip overlay is constructed by taking into account the joining time of nodes, and hence the part of the stream they should be receiving, along with the size of their buffer, to maximise the chance of overlap between gossiping nodes. The data exchange proper uses digests and pull operations.

Unstructured Epidemic Multicast [51] proposes a framework to combine an eager and a lazy push approach (punctual and lazy dissemination in our terminology) to multicast messages in a way that approximates structured multicast-protocols. The key idea consists in preferably selecting "good" nodes (according to some metric, e.g. bandwidth, latency, etc.) to eagerly propagate gossip messages, while falling back onto a lazy dissemination approach for other nodes.