Transparent Componentisation: High-level (Re)configurable Programming for Evolving Distributed Systems

Shen Lin[1], François Taïani[1], Marin Bertier[2], Gordon Blair[1],Anne-Marie Kermarrec[2]
[1] Lancaster University, Lancaster, UK
{s.lin, taiani, gordon}@comp.lancs.ac.uk
[2] INRIA/IRISA, Rennes, France
{marin.bertier,anne-marie.kermarrec}@irisa.fr

ABSTRACT

Component frameworks and high-level distributed languages have been widely used to develop distributed systems, and provide complementary advantages: Whereas component frameworks foster composability, reusability, and (re)configurability; distributed languages focus on behaviour, simplicity and programmability. In this paper, we argue that both types of approach should be brought together to help develop complex adaptive systems, and we propose an approach to combines both technologies without compromising on any of their benefits. Our approach, termed *Transparent Componentisation*, automatically maps a high-level distributed specification onto a underlying component framework. It thus allows developers to focus on the programmatic description of a distributed system's behaviour, while retaining the benefits of a component architecture. As a proof of concept, we present WhispersKit, a programming environment for gossip-based distributed systems. Our evaluation shows that WhispersKit successfully retains the simplicity and understandability of high-level distributed language while providing efficient and transparent reconfigurability thanks to its component underpinnings.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Distributed Programming

General Terms

Design

Keywords

distributed language, component, reconfiguration, gossip

1. INTRODUCTION

Both component frameworks and high-level distributed languages have emerged as powerful technologies to realise distributed systems. Component frameworks allow developers to assemble systems from reusable components that explicitly expose their operational dependencies as interfaces and receptacles (i.e. provided and required services). They thus encourage a compositional approach to system constructions that fosters modularity, reuse, and configurability. They also facilitate the development of dynamically adaptive systems: knowledge about interfaces and receptacles allows the reconfiguration logic to reason about dependencies, while dynamic bindings provide a simple mechanism to update a system at runtime. These benefits make components a particularly popular approach to develop middleware platforms. They have been successfully applied both in the industry (c.f. EJB, CORBA Component Model, DCOM), and in middleware research, giving rise to lightweight component technologies (OpenCom [5], Fractal [3]) and their associated middleware frameworks (GridKit [6], RAPIDWare [17]). However, because component frameworks focus on structure rather than algorithms, they are limited in their ability to represent a system's detailed behaviour, such as execution flows and message exchanges [4]. This in turn makes it difficult for developers who are unfamiliar with a particular framework to understand its logic [7].

In contrast to components, high-level distributed languages focus explicitly on the algorithmic logic of distributed systems rather than on their compositional structures. Examples include protocol specification languages (e.g. Lotos [21], Estelle [1], PLAN-P [20], Promela++ [2], Mace [13]), declarative networking such as P2 [16], and recent macroprogramming languages such as Kairos and Regiment that allow developers to program a distributed system as a single entity (i.e. macro level) [9, 18]. Compared to components, however, these high-level distributed languages make it more difficult to reason about and implement dynamic changes, because they do not explicit expose fine-grained dependencies between collaborating program parts. As a result, they often require adaptive behaviours to be statically hard-wired prior to system deployment, or a complete new version to be installed to replace an existing one.

Both component frameworks and high-level distributed languages are highly complementary, yet they are difficult to combine: Bringing components to high-level distributed languages tend to undermine their programmatic simplicity by forcing developers to navigate back an forth between structural and behavioural concerns. This tension becomes worse with as the structural decomposition becomes finer, further limiting this approach for fine-grained adaptation. Symmetrically, adding a behavioural dimension to compo-

SAC'11 March 21–25, 2011, TaiChung, Taiwan

nents makes composition harder to grasp and manipulate by developers, canceling the very benefits it should bring.

In this paper, we propose to combine the elegance of high-level distributed languages and the structure of component frameworks without losing on any of their benefits. We propose to do so by transparently transforming a high-level distributed program into a fine-grained component architecture, an approach we have termed *Transparent Componentisation*. In doing so, we allow developers to focus solely on the behavioural logic of their distributed system, while still benefiting from the reusability and (re)configurability brought by components. Transparent Componentisation provides three key benefits towards the development of distributed systems: (i) by providing high-level yet simple expressions, it helps developers *reason* about a system's logic; (ii) by relying on an underlying component framework, it allows developers to *reuse* the domain knowledge captured by such frameworks, without mastering all their intricacies; (iii) by automatically maintaining a mapping between high-level expressions and their underlying component realisation, it enables smooth and highly efficient *reconfiguration strategies*.

In the remainder of this paper, we first motivate the need for Transparent Componentisation and present related work (section 2). We then expose the general principles of our approach (section 3), before presenting a proof of concept, WhispersKit, in the particular context of gossip-based overlays (section 4). We then show how it retains the simplicity of high-level distributed languages, while offering the benefits of component frameworks, in particular for dynamically evolving systems (section 5). We finally conclude in section 6.

2. BACKGROUND AND MOTIVATION

2.1 Lightweight Component Frameworks

Contemporary distributed systems are increasingly heterogeneous and dynamic. They often involve nodes with varying capacities, and operate over a large spectrum of networking technologies (e.g. fixed networks, mobile ad hoc networks, satellite links, etc.). They must cater for highly *dynamic* operational conditions: nodes may move, may join or leave, may fail; network conditions may evolve abruptly. The resulting complexity requires systems to provide customised services adapted to each of these *heterogeneous* operating environments. Furthermore, these systems often need to adapt to changing requirements and environments by reconfiguring themselves at runtime. The need to cater for different requirements and conditions, and to adapt when these change, can be observed in a wide range of distributed application domains such as mobile adhoc routing [19], service discovery [8], and gossip-based overlays [14].

To address such heterogeneity and dynamicity, fine-grained reflective component frameworks have been successfully proposed to develop distributed systems in the past [19, 8, 14]. In these approaches, the key elements of a distributed system are implemented as reusable software components. These components can be flexibly composed to form various customised systems that perform optimally on different operating contexts. Reflection provides introspection and intercession mechanisms that facilitate adaptation to changing conditions. Components also allow a fine-grained strategy to the construction of adaptive systems, avoiding switching entire implementations. Finally, the use of components fosters sharing and reuse across systems, thus reducing both development effort and resource overhead.

2.2 High-level Distributed Languages

Designing and implementing robust and efficient distributed applications is generally considered a difficult and error-prone task [13]. In response, researchers have proposed several high-level protocol specification languages to help realise distributed systems. Examples include Lotos [21], Estelle [1], PLAN-P [20], Promela++ [2], Mace [13]. These languages tend to describe distributed algorithms as state machines that operate on individual nodes. In this model, each node updates its local state and triggers local tasks (e.g. transmitting messages, delivering application data) as a reaction to specific events such as network messages, timers, internal interactions, and user commands.

Protocol specification languages typically offer simple yet concise expressions to describe distributed and parallel algorithms. Their formal specifications also enable them to automatically validate the logical consistency and protocol correctness against safety requirements provided by protocol developers. Despite these common benefits, individual protocol specification languages tend to support system development in different aspects. Estelle and LOTOS are formal specifications that capture the essence of many distributed protocols and algorithms, but abstract away from low-level implementation details such as security, reliability, and efficiency. Promela++ is an extension of C that provides optimisation mechanisms to automatically generate efficient C code, while allowing the use of the SPIN model checker for verification. PLAN-P is similar to Promela++, but it uses a runtime interpreter to execute its high-level language expressions while achieving the same efficiency as compiled Java code. Mace provides expressions such as events and transitions to explicitly describe a distributed algorithm as a reactive state transition system.

More recently, macro-level programming has been brought forward, in particular in the area of wireless sensor networks [9, 18] to simplify the specification of distributed algorithms. Macro-level programming considers a distributed system as a single entity, and automatically translates a system-level program into per-node processes. Thus, macro-programming is able to concisely express how data flows over a network (i.e. network level or macro level), and saves the programmer from explicitly dealing with network messages used to access remote data.

2.3 Analysis

As discussed above, component frameworks and high-level distributed languages have both been successfully applied to build distributed systems. Unfortunately, both types of approach offer benefits that to a large extent do not overlap. Rather, the strengths of one category of approaches are often the weaknesses of the other.

Because component frameworks focus on structure rather than behaviour, their architectures might not be straightforward or appealing to domain experts (i.e. researchers and developers) that focus on inventing new protocols or improving existing ones. The fine-grained component frameworks advocated for highly adaptable systems require a substantial additional development effort, which in turn might further limit their interest to practitioners.

In contrast to component framework, high-level distributed

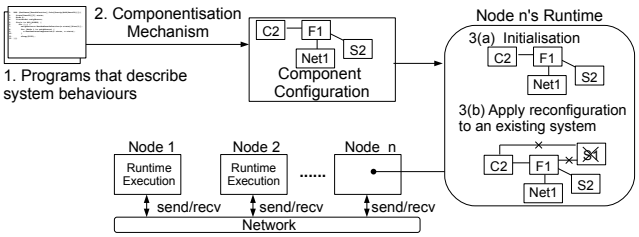

Figure 1: Steps of Transparent Componentisation

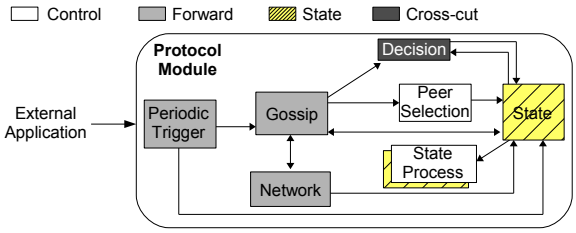

Figure 2: GossipKit's common architecture

languages offer simple yet concise expressions to describe distributed algorithms. They thus allow protocol developers and maintainers to focus on the system's logical behaviour rather than on its architectural abstractions. Many of these languages also provide some form of modularity to structure the resulting programs (e.g. event handlers, protocol stack composition). However, such structures typically do not explicitly capture the dependencies that might exist between the various parts of an algorithm, which limit their applicability to dynamic reconfiguration. When applied at a fine-grained level, they also force developers to navigate back an forth between structural and behavioural concerns, a tedious and possibly error-prone demand on the attention of developers [10].

To combine the strengths of both perspectives, this paper proposes an approach we have termed *Transparent Componentisation* to support the development of adaptive distributed systems. This approach separates logical concerns (i.e. what a protocol does) from structural ones (i.e componentisation), while automatically mapping logic unto structure. It thus allows developers to focus on the algorithmic aspects of their systems while enjoying minimum levels of structural scattering. Thanks to the automatic mapping of the program's logic onto an underlying component framework, the leg-work involved in fine-grained componentisation occurs the surface, in a fully transparent manner.

3. TRANSPARENT COMPONENTISATION

3.1 Overview

Transparent Componentisation involves three key steps (Figure 1):

1. The behaviour of a system is specified in a high-level distributed language as a set of program files.
2. The programs are then processed by a component mapping mechanism to generate a corresponding component configuration.
3. This component configuration is fed to the runtime environment of a component framework that operates on individual nodes of a distributed system.

If the system is just being initialised, the underlying component framework loads and connects components according to the passed configuration (phase 3a in figure 1). If the system is already running on a previous configuration, a *reconfiguration manager* compares the new configuration against the current one and modifies the running system using a sequence of reconfiguration operations such as replacing components and rebinding connections (phase 3b).

3.2 Proof of Concept

To demonstrate the practical value of Transparent Componentisation, we present WhispersKit, a macro-programming tool chain geared towards gossip-based overlays. WhispersKit comprises the Whispers language and the GossipKit component framework. Developers specify a system's behaviour using Whispers, and this specification is then automatically mapped onto per-node GossipKit configurations.

Gossip Protocols provides highly scalable communication in large-scale networks. They spread information in the way a rumour is randomly gossiped amongst humans [12]. Gossip protocols are available for a wide range of services (e.g. aggregation, membership, broadcast), over various network types (e.g. IP-based and MANETs). They are also typically collaborative, as multiple protocol instances may collaborate to achieve more sophisticated services [12, 15]. They thus often need to be customised for a targeted application domain and operating environment, and are a good representative example of a protocol family that can benefit from a component-based implementation.

GossipKit [14] is an existing component-based middleware framework for (re)configurable gossip overlays. Its underlying design is based on the observation that most gossip protocols can be decomposed into a common component architecture (figure 2). In GossipKit's architecture, the `Gossip` element is responsible for forwarding gossip messages using the transport mechanism(s) provided by the `Network` element. Gossip messages are disseminated to peers selected by probabilistic algorithms captured in the `Peer Selection` element. The message dissemination performed by the `Gossip` element can either be triggered reactively by an external application or periodically by the `Periodic Trigger` component. The `State` element maintains the data (e.g. a sensor reading, a buffer of network packets, or a neighbour list) that is gossiped between nodes and is updated by the `State Process` element. Finally, `Decision` captures the conditions for executing these functional elements.

Each element in figure 2 can be implemented by a single GossipKit component or by multiple such components. These components are then composed together according to an XML-based configuration file, which also forms the basis to compose several gossip protocols together.

The Whispers Distributed Language supports macro-programming style expressions to simplify the description of gossip-based overlays. Furthermore, Whispers programs can be automatically translated into concrete GossipKit component systems that support dynamic (re)configuration. To achieve both simplicity and component reuse, WhispersKit relies on *a controlled mapping* of the expressions of Whispers onto GossipKit's component model. In some

cases, WhispersKit allows highly reusable components to be directly referenced in the Whispers language as invokable entities. For instance, component types such as `Peer Selection` and `State Process` in figure 2 fall into this category. In most other cases, parts of a Whispers program, including expressions that describe execution sequences, control flows, and message exchanges, are captured as traditional programmatic constructs, to support the programmability and the understandability of the language. These constructs do not have one-to-one counterparts in the GossipKit component model but are instead synthesised into components following an automated analysis of the program.

Finally, because a gossip system typically involves multiple collaborative protocols, Whispers provides high-level expressions to describe the interactions between gossip protocols. These expressions can then be automatically transferred into collaborations of coexisting gossip protocol instances in a concrete component system.

Implementation WhispersKit is implemented as a tool chain supporting the key steps of figure 1. This tool chain comprises a compiler that parses a Whispers specification and generates a GossipKit architecture. The outcome of a compilation is a configuration file that describes how the generated components should be composed with pre-existing GossipKit components to realise the original Whispers specification. Finally, this configuration file is deployed together with the GossipKit components onto the GossipKit runtime of individual nodes to initialise or to reconfigure a gossip-based application. WhispersKit' implementation consists of about 800 lines of JavaCC code and 4000 lines of Java for its compiler, and 3400 lines for GossipKit. WhispersKit' source code can be downloaded online[1], along with eight gossip overlays specified in Whispers.

3.3 Case Study

As an example, we illustrate WhispersKit on the random peer sampling (RPS) protocol [12]. RPS maintains a random overlay topology of a communication group. To do so, each node maintains a "random sample" that contains the identifiers of C random peers in the group (C is a small constant number). The RPS protocol runs periodically, and at each period a node n selects a randomly peer i from its random sample, and sends a copy of n's random sample to i. On receipt of n's random sample, node i immediately replies with a copy of its random sample. On receiving another node's sample, each node first merges its local random sample with the remote one to form a temporary sample with size $2C$, and then discards C random peers from the temporary sample to obtain the updated sample with size C. In doing so, RPS allows each node in the communication group to obtain a fresh random sample of the group membership at a regular rate.

The code of RPS in Whispers is shown in figure 3. In Whispers, the entire declaration of a protocol's behaviour is enclosed in a protocol block (between line 1 and 10) that starts with a protocol identifier (here `RPS`) so that external protocols may access its data or services to realise composite protocols. The first section in a protocol block is a variable declaration section (lines 1-2), where protocol states and node variables are declared. Line 2 declares a `sample` state variable, which is a list of `Node` objects with size 5. Line 3 declares two `Node` variables (`n` and `i`).

[1]http://www.lancs.ac.uk/postgrad/lins6/GossipKit.html

```
1. RPS {
2.  State sample = new State[Node:PeerID][Size=5];
3.  Node n, i;
4.  every (5000) { // do the following every 5000 ms
5.    foreach(n in getAllNodes()) { // for each node n
6.      i=n.RandomPeerSelection(n.sample)[Size=1];
7.      n.sample.add([n]);
8.      i.RandomCompress(i.sample,n.sample)[Size=5];
9.      n.RandomCompress(i.sample,n.sample)[Size=5];
10.}}}}
```

Figure 3: Whispers program of the RPS protocol

The actual protocol behaviour, which executes every 5000 milliseconds, is specified in a `every` block between line 4 and 10. In this block, lines 5 and 6 initialise the node variables `n` and `i`. Node `n` is initialised through a distributive statement `foreach (n in ...)` at line 5. Distributive statements are key to Whispers's handling of distribution, in that they anchor the locus of computation on a local node (here `n`), and provide a reference point to which other variables (such as the neighbours of `n`) are defined. Line 6 initialises `i` to a random neighbour of `n`.

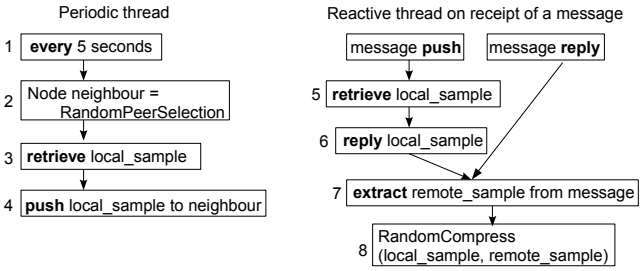

Figure 4: Per-node program of RPS

Lines 7-9 capture the core of the RPS algorithm. Each node `n` fist adds itself to the sample state about to be disseminated (Line 7), to avoid from being isolated from other peers [12]. Node `n` and `i` then exchange their respective sample states, and each create a new state by merging the received state with their current one (lines 8-9). These two lines illustrate a key difference between Whispers' macro-level programming approach and more traditional per-node programming: macro-level programming describes nodes that access each other's data as if they were operating in the same memory space of a single program, thus enabling interactions between nodes to be described without explicitly handling any messages. More precisely, line 8 specifies that node `i` should obtain `n`'s sample state, and then apply the `RandomCompress` function on its own state and that of node `n`. In line 8, `RandomCompress()` is invoked on node `i` (i.e. `i.RandomCompress()`), after `i` has received `n`'s sample state through the network. Similarly, line 9 indicates that node `n` receives `i`'s state as a reply, and then directly applies the `RandomCompress` function on its own state and that received from node `i`.

When presented with a Whispers program such as in figure 3, WhispersKit generates an abstract per-node program. This per-node program, sketched in figure 4, explicitly sends and receives messages on each local node and handles the node's reaction to the reception of each type of messages. Compared with the simple macro-level program of Figure 3, a per-node program explicitly captures low-level programming details such as threading, message handling, data synchronisation, remote data access, and network in-

terface management for sending or receiving messages. As part of Transparent Componentisation, the outcome of this translation is not visible to developers, but used to generate the concrete component architecture that will be deployed on each node.

To generate this architecture, WhispersKit analyses the per-node program of RPS, and maps the statements in the per-node program onto appropriate GossipKit components and their connections. For instance, the periodic behaviour is mapped onto the `Periodic Trigger` component that is configured to execute every 5 seconds; the two types of message transmissions used in the RPS protocol (push and reply) are mapped onto two different configurations of the `Gossip` component. These mappings result in the component configuration of figure 5, which can be used by GossipKit to initialise the RPS system.

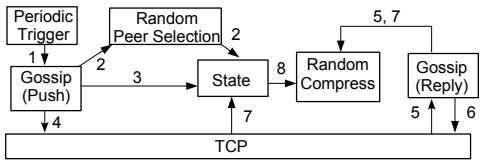

Figure 5: Component realisation of RPS

4. EVALUATION

We have evaluated Transparent Componentisation according to two main criteria: *simplicity*, i.e. can Transparent Componentisation provide a *simpler* and *more understandable* way to describe distributed algorithm compared to a raw component approach?; and *reconfigurability*, i.e. can Transparent Componentisation retain the benefits of componentisation, such as *reconfigurability*? The following presents the evaluation of WhispersKit in terms of these criteria, based on our implementation of eight gossip protocols (including the RPS protocol presented in section 3.2). To provide a baseline for comparison, each protocol was implemented three times: first with WhispersKit, then using GossipKit only, and finally directly from scratch in Java.

Simplicity As an indirect measure of simplicity, a property particularly difficult to assess, we compare the lengths (in lines of code) of the three different implementations of all eight gossip protocols (in Java, GossipKit component configuration, and Whispers). The result in table 1 shows that Java implementations typically require between 270 and 540 lines of code, while Whispers reduces this programming effort to no more than 15 lines for all protocols. Finally, Whispers also requires less lines of code to specify a gossip system, comparing with GossipKit's XML-based configuration language. Although the difference in this comparison is not as significant as the comparison with Java, our experience is that the style of Whispers's specification is much simpler and more understandable than that of GossipKit's configuration language. This is mainly because Whispers is able to directly describe a protocol's behaviour such as execution sequences, control flows, and message propagation, whereas GossipKit's configuration language requires 7.8 components, 10.3 parameter settings, and 8.4 connections on average to describe a gossip protocol, which is relatively more complex to program and understand.

Reconfigurability As an illustration of the benefits that WhispersKit obtains from componentisation, we present

Approach	Minimum	Maximum	Average
Java	277	544	432
GossipKit	42	60	52
Whispers	17	20	19

Table 1: The min, the max, and the avg number of lines of code used to implement the 8 gossip overlays

a simple experiment of dynamic reconfigurability. The scenario is an evolving distributed system that needs to change the overlay topology it maintains to support different high-level applications. This scenario involves three WhispersKit programs and two sequential reconfigurations. The target system is made of 100 nodes deployed in a 10×10 grid, and uses the Jist/SWANS simulator[2] to simulate the underlying TCP/IP network.

Initially all nodes run the RPS protocol [12] to maintain a random graph for peer sampling (the 1st snapshot in figure. 6). The first reconfiguration consists in launching an implementation of T-Man [11] to construct a *ring* topology. Because T-Man relies on RPS to sample peers, the WhispersKit compiler will include RPS in the configuration it generates for T-Man, and the reconfiguration script will insure the components required by T-Man get instantiated on top of the running RPS. The reconfiguration script is then triggered on node 0, propagates through the network, and eventually converges to the ring topology (the 2nd and the 3rd snapshots in figure. 6). Once the ring topology has been reached, a second reconfiguration is triggered to use a second implementation of T-Man to build a *grid* topology (the 4th and the 5th snapshots in figure. 6).

This experiment demonstrates WhispersKit' ability to support reconfiguration at different levels of granularity. The first reconfiguration is coarse-grained: it deploys an entirely new protocol (i.e. T-Man for constructing a ring topology) atop RPS, injecting 8 new components and 10 new bindings. The second one is fine-grained, and only involves the `State Process` component of T-Man and two bindings. Thanks to WhispersKit' Transparent Componentisation, developers do not need to worry how coarse- or fine-grained a reconfiguration is in practice, or which architectural dependencies should be taken care of. These concerns are automatically addressed by the underlying tool chain, on the sole basis of the intended high-level Whispers specifications.

Efficiency Although not central to this study, we have examined the runtime performance of our approach. To measure the overheads incurred by our approach, we measured the average local processing time of one gossip round for each version of the eight protocols. The results show that both the GossipKit and Whispers versions run substantially slower than any direct Java implementations due to the inevitable overhead of component invocation. However, the overheads they incur (0.5 ms on average) are still much smaller than the network latencies encountered in wide-area networks (typically from tens to hundreds of milliseconds), and comparable to that of local-area networks (typically a fraction of millisecond). The result also shows that the gossip systems implemented by using WhispersKit do not introduce any extra overhead over GossipKit's runtime. This is understandable since the result of a Whispers compilation is a GossipKit component configuration.

[2]http://jist.ece.cornell.edu/

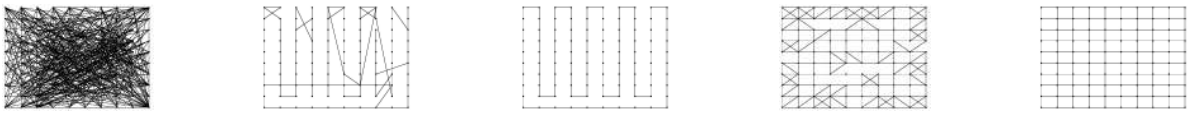

Figure 6: Evolving overlay topologies achieved by runtime protocol reconfiguration: 1) initial random graph maintained by the RPS protocol, 2) start constructing ring topology after the first reconfiguration, 3) ring topology converged at the 16th gossip round, 4) start constructing grid topology after the second reconfiguration, 5) grid topology converged 12 gossip rounds after the second reconfiguration

5. CONCLUSION AND FUTURE WORK

This paper has presented Transparent Componentisation, a new programming methodology to support the development of (re)configurable distributed systems. Transparent Componentisation provides an automatic mapping between a high-level protocol specification and a underlying component framework. It thus allows developers to focus on the specification of the logical behaviour using simple yet concise expressions (e.g. macro-programming style). Meanwhile, it retains the benefits of a fine-grained runtime component architecture such as reusability and (re)configurability in a transparent manner. As a proof of concept, we have presented the WhispersKit framework for programming (re)configurable gossip systems, and discussed a preliminary evaluation of WhispersKit based on the implementation of eight gossip protocols within WhispersKit.

WhispersKit demonstrates the feasibility of our approach and opens up interesting avenues for future research. First, WhispersKit is potentially extensible to support a wider range of component-based distributed applications beyond gossip-protocols. Second, GossipKit's ability to express and support coexisting protocols raises the issue of both the interferences and synergies between multiple protocols running in the same infrastructure [15]. Properly extended, WhispersKit can probably help in addressing these issues, and we plan to look at this in more detail.

6. 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).

7. REFERENCES

[1] P. Amer and F. Ceceli. Estelle formal specification ofiso virtual terminal. In *Computer Standards and Interfaces Conference*, 1990.

[2] A. Basu and M. Hayden. A language-based approach to protocol construction. In *Domain Specific Languages Workshop*, 1997.

[3] E. Bruneton, T. Coupaye, and M. Leclercq. An open component model and its support in java. In *7th International Symposium on Component-Based Software Engineering*, 2004.

[4] P. Clementse. A survey of architecture description languages. In *8th International Workshop on Software Specification and Design*, 1996.

[5] G. Coulson, G. Blair, and P. Grace. Opencom v2: A component model for building systems software. In *Software Engineering and Applications*, 2004.

[6] G. Coulson, P. Grace, and G. Blair. Gridkit: Pluggable overlay networks for grid computing. In *Distributed Objects and Applications Conference*, 2004.

[7] G. Edwards, G. Deng, and D. Schmidt. Model-driven configuration and deployment of component middleware publisher/subscriber services. In *Generative Programming and Component Engineering Conference*, 2004.

[8] C. Flores-Cortés, G. Blair, and P. Grace. An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments. In *Dist. Sys. Online*, 2007.

[9] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *DCOSS*, 2005.

[10] M. Hiltunen, F. Taïani, and R. Schlichting. Reflections on aspects and configurable protocols. In *5th international conference on aspect-oriented software development*, pages 87–98, 2006.

[11] M. Jelasity and O. Babaoglu. T-man: Gossip-based overlay topology management. In *Engineering Self-Organising Systems*, 2005.

[12] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Steen. Gossip-based peer sampling. In *ACM Trans. Comput. Syst.*, 2007.

[13] C. Killian, J. Anderson, and R. Braud. Mace: Language support for building distributed systems. In *Programming Language Design and Implementation*, 2007.

[14] S. Lin, F. Taiani, and G. Blair. Facilitating gossip programming with gossipkit framework. In *Distributed Applications and Interoperable Systems*, 2008.

[15] S. Lin, F. Taiani, and G. Blair. Exploiting synergies between coexisting overlays. In *Distributed Applications and Interoperable Systems*, 2009.

[16] B. Loo, T. Condie, and et. al. Declarative networking: Language, execution and optimization. In *SIGMOD*, 2006.

[17] P. McKinley. Experiments in composing proxy audio services for mobile users. In *Distributed Systems Platforms Conference*, 2001.

[18] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *6th international conference on Information processing in sensor networks*, 2007.

[19] R. Ramdhany, G. Coulson, and P. Grace. Manetkit: Supporting the dynamic deployment and reconfiguration of ad-hoc routing protocols. In *Middleware Conference*, 2009.

[20] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *Symposium on Reliable Distributed Systems*, 1998.

[21] P. van Eijk and M. Diaz. *Formal Description Technique Lotos*. Elsevier Science Inc., 1989.