Lightweight Service-Based Software Architecture

Mikko Polojärvi and Jukka Riekki

Intelligent Systems Group and Infotech Oulu
University of Oulu, Oulu, Finland
{mikko.polojarvi,jukka.riekki}@ee.oulu.fi

Abstract. This article describes the Simple Event Relaying Framework (SERF), a novel service-based software architecture designed especially for resource constrained settings and facilitated cooperation between applications. The proposed prototype framework aims at utilizing publish/subscribe and peer-to-peer, technologies that are usually encountered only in higher level inter-device networks, inside the software architecture of individual applications in very simple form. The idea is to introduce a simple core architecture on which more advanced features can be built on. The research will be conducted primarily by creating prototype solutions to real life problems and learning from the experience.

1 Introduction

One trend in information technology nowadays is that modern mobile devices are more and more composed of larger number of small applications. One smart phone might easily contain over one hundred user-installed applications. We can consider the applications networked to some degree, but often they are still quite separated from each other. They seldom communicate or cooperate with each other, and even if they do, it is usually done only in the way the application developers have specifically envisioned.

This does not need to be limited to the software inside a single mobile device. For example, in Smart Space research everyday environment is envisioned as full of networked smart objects, or miniature computers or sensors that fulfill a very specific purpose, for example detecting the user's presence in a room. This information alone is not useful alone, it needs to be combined with an application that can use the information in some way, for example by turning the lights on. In the same way as above, it is neither possible for the smart object designers to know all the ways the information could be utilized, nor possible for the application designers to know all the future devices that could provide the information. This is a problem that limits the usefulness of the smart space applications and objects.

We could argue same applies to individual applications as well. Applications usually consist of a number of smaller software components, and if we consider memory references, function calls etc. as links, we can consider the software components networked. While running, these components do lot of information processing. If the application developer cannot think of a reason to expose the results of this processing outside, the results probably will not be available to other applications, since building for such flexibility usually also consumes resources.

Therein lies great potential for development. If software components could be made to open up more efficiently for cooperation with each other, it would open the way for using applications in many ways not originally planned for. In order to allow such unplanned cooperation between applications, the internal state of the applications needs to be implicitly exposed outside the application.

One solution could be to make even the smallest component loosely coupled and communicate using the publish/subscribe paradigm. Each component would have clear cut purpose in the application and depend very little or not at all on other components in order to fulfill the purpose. The components would send (i.e. publish) messages around so that only the components that have expressed interest (i.e. subscribed) in a certain message type end up receiving them. Senders would not care who they are sending to, and receivers would not care where they get the messages from, as long as the content of the message is interesting. As a result, senders and receivers could be freely added, removed, moved or modified without breaking the whole system, allowing the components to cooperate with each other in unforeseeable ways.

Naturally, within the idea there are many challenges, but first and foremost is the question of how to organize the communication between the components in practice? The QoS requirements for communication at different levels (inside an application, between applications, between devices etc.) vary greatly. There already exist numerous solutions utilizing publish/subscribe, but most existing solutions are aimed at large scale inter-device networks and often assume a single mobile device as the smallest unit in the network of components. Many publish/subscribe solutions are simply too heavy computationally for use in lower level communications, services and algorithms. This is further aggravated by the fact that in mobile devices, power, storage and computing resources are considerably limited. Other challenges are discussed later.

This is where my doctoral thesis research comes in. The research investigates the possibility of utilizing publish/subscribe in considerably lower level settings, with the aim of allowing unplanned cooperation between components. Towards this purpose I have developed a novel component based general purpose software framework prototype called the Simple Event Relaying Framework (SERF). SERF is a continuation of my earlier work, the Ideasilo framework published in [1]. This particular prototype concentrates in the software architecture inside individual applications. In this paper I will explain the detailed design, experiences received thus far, research methods and expected results of my work. But first, I will present a review about other related solutions.

2 Related Work

Service Oriented Architecture (SOA) is one prominent modern tool utilizing loosely coupled modules. In fact, the current SERF prototype has been largely inspired by the principles of SOA [6]. Most of the principles have been adopted unchanged or in slightly modified form. However, some principles have been discarded in order to accommodate usage in resource constrained settings.

Android's Intent system [3] is also meant to facilitate inter-component cooperation within a single mobile device. An intent is a passive data structure holding an abstract definition of an operation to be performed or a description of something that has happened. In case a component interested in the service is not loaded by the time the intent is executed, the system automatically instantiates them.

Among experimental solutions, Network on Terminal Architecture (NoTA) [4], developed in the Nokia Research Center and first released in 2005, bears some similarities with SERF. Both are have a similar purpose (facilitate software development) and solution principle (use SOA in device level settings). The Smart-M3 [5] project is similar in the sense that it also aims at software cooperation at multiple levels, but concerns itself mostly on inter-device part. Both NoTA and Smart-M3 differ greatly from SERF in design, and neither use publish/subscribe in messaging by itself.

3 Proposed Solution

As stated before, SERF is designed for investigating the possibility of utilizing publish/subscribe also within low level software components. The current prototype concentrates primarily in resource-wise the most demanding part, communication between components inside individual applications, while keeping a secondary focus on inter-application and inter-device communication. Therefore the design concentrates in providing the communication framework with as few resources as possible.

SERF addresses the challenges of designing general purpose software frameworks by making very few assumptions about the used technologies or applications built upon it, and not trying to do too much. In fact, in my approach SERF aims at providing just a conceptual answer the question: "which message should be delivered to which components?"

The SERF research is based on the following hypotheses:

1. Working and useful applications can be formed using primarily loosely coupled components with less resources than creating every component from scratch.
2. Higher level framework functionality can be implemented as applications on top of a very simple core framework.
3. It is possible for different applications to cooperate in new ways that have not been taken into account at design time.
4. Lightweight messaging architecture can offer significant reductions in computational workload as compared to more feature-rich solutions.

The design of the first prototype of SERF is described in detail at [2]. Although the current solution features an upgraded scheduling solution, most of the design principles and implementation still apply to the current prototype. A brief summary of the design is presented next. The main principles of the current approach can be summarized as follows:

1. The framework is kept as simple as possible.
2. The main idea of the framework is conceptual and platform-independent.
3. An application is composed of a many small components that send messages to each other.
4. The components form an acyclic network with each other.
5. The framework will route messages from the sender to the recipient(s) by the topic of the message. The framework will not concern itself with the identity of the sender of the recipient.
6. Messages will not describe what should happen in the application. Instead, the messages simply describe what is happening at the moment. Other components may then decide how to act on the information.
7. The framework concentrates on keeping the overhead of routing the messages from senders to receivers at minimum.

Figure 1 presents an example about the current component structure. The structure consists of event processors, routers and filters, and thread schedulers. Event processors ($P_1 - P_5$ in the figure) can be understood as the application components mentioned earlier. These are developed by the application designers using the framework. Real applications are formed by having them communicate with each other. They are basically pieces of code that generate and receive messages, the framework being responsible for executing the event handling code when necessary.

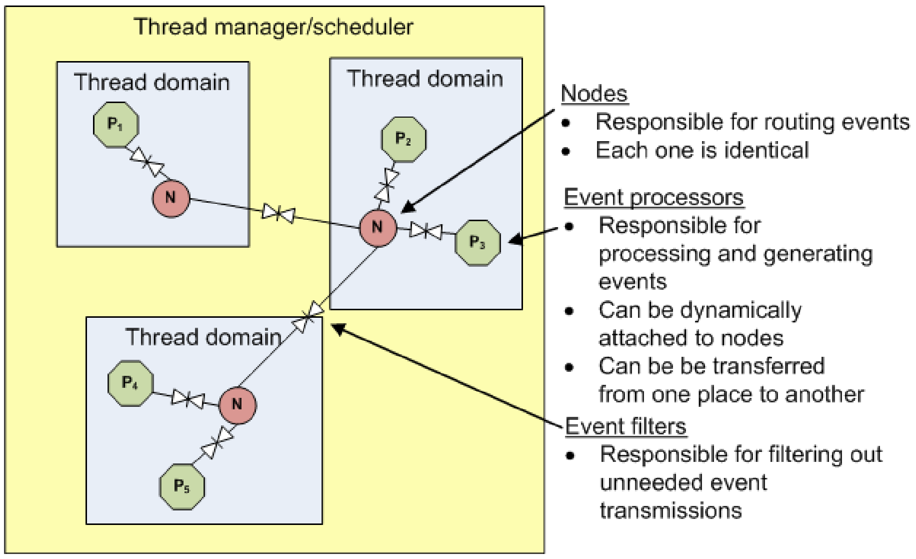
Thread manager/scheduler
Thread domain
Thread domain
Thread domain
P1
P2
P3
P4
P5
N
N
N
Nodes
• Responsible for routing events
• Each one is identical
Event processors
• Responsible for processing and generating events
• Can be dynamically attached to nodes
• Can be be transferred from one place to another
Event filters
• Responsible for filtering out unneeded event transmissions

Fig. 1. SERF routing example

Routing nodes (the N:s) in figure 1 are responsible for routing the messages to processors and other routing nodes. These are all identical to each other, although their configuration may differ. When necessary, the thread scheduler (the surrounding big box) assigns a thread to work at each node demanding attention. The thread routes messages, executes processors and returns to wait for more work on other nodes. The solution ensures that all processors attached to the same node are always executed in the same thread (as indicated by the Thread Domain box around each node). One important aspect of this solution is that this kind of structure supports multi-processor architecture in the sense that it encourages dividing the application into clear cut components. Each component can be easily assigned to be worked on by a separate processor core, and the number of worker threads can be easily adjusted. The event-based messaging design also inherently reduces many concurrency problems usually associated with thread-based programming.

The event filters (the diode signs) in figure 1 are responsible for handling the actual routing of messages. Each link from router to router or router to processor contains two filters, one for each direction. Every message contains meta information about the topic of the message, where a single topic is represented with a single bit in a particular position in an array of bytes. This way the demands of a complete branch of nodes in the network can be represented in compact form with byte arrays and evaluated efficiently with binary operators. These filter evaluations are repeated once for each link in a routing node. Because in SERF the routing node network cannot contain loops, this potentially results in larger logical distance between the nodes as compared to cyclic networks. However, if the overhead of a single filter check and event transmission can be kept sufficiently low, the overhead may still be less than by using more complex routing solutions.

Obviously a single bit cannot carry much information, just whether a particular topic applies to the event in question or not. What a particular topic means is not modeled in any way in the framework. Instead the framework relies on the designers agreeing on the significance of each given topic. An important point here is that this solution is essentially a tradeoff of features for performance. It is not intended to be perfect solution for every given situation in itself. There are indeed many cases where this kind of messaging cannot be considered sufficiently scalable, robust or flexible. However, we hypothesize that this kind of solution is still able to support a layer of services offering widely enhanced features, built on top of the very simple core framework. How this can be done efficiently is another promising topic for research.

An important notion here is that although the current SERF prototype provides a simple key-value structure for the content of the messages, SERF actually does not assume any particular structure for the contents. Therefore the application designers are free to use any structure that can be transformed into serial form, although communicating components will need to know the structure in order to be able to access the contents.

4 Methods

In the next phase of my research I intend to evaluate the possibility of utilizing publish/subscribe in low level settings by searching answers to the following research questions:

1. What kind of advantages can be achieved by applying lightweight publish/subscribe in low-level settings?
2. What disadvantages does the above have? In order to achieve the benefits, what compromises must be made?
3. On what kind of application areas will the advantages outweigh the disadvantages?

In order to provide the answers, my approach is primarily based on experimenting with prototypes with the aim of gaining broadened understanding on using SERF in practice. The plan is to iterate the following workflow:

1. Based on literature review on the theory of software frameworks and messaging, and the experiences gained from the previous steps, develop and optimize the SERF framework.
2. Discover a real life problem where the current SERF framework could be utilized to a desirable effect. Here the choice of the problem area does not need to be limited only to resource constrained settings, as it is valuable to study how well SERF performs also in less resource constrained settings.
3. Study the theory related to the problem area at hand and different kinds of possible solutions.
4. Develop a prototype solution to the aforementioned real life problem using SERF and software components required by the application, preferably ones already created earlier in the workflow. Also, when possible, a second solution using another feasible competing solution should be implemented for comparison.
5. Evaluate the prototype quantitatively by benchmarking and qualitatively by user tests, compare the results to competing solutions and publish the findings.

This plan provides good opportunities for interdisciplinary research collaboration. Moreover, the workflow itself serves as a sanity check on the main hypotheses presented earlier. The workflow also works as an experiment on how easily software components made earlier in the workflow can be utilized in later projects without essentially modifying the components and without knowing what a given component is going to be used for later.

As SERF is still in prototype stage, there are still many possibilities for further improvement. A fundamental question here is deciding which improvements should be implemented in the framework itself and which in the application layer, and which improvements rejected outright. Among others, the following possibilities will need consideration:

1. Similar to Android, the framework could support instantiation of event routers and processors when needed.
2. The framework could support mobile software components, i.e. pieces of code that are sent to remote devices for execution.
3. The event filtering solution could be still improved. Especially, utilizing Bloom filters [7] instead of byte arrays in the filtering solution could offer performance advantages.

5 Expected Results

There are many challenges associated with intra device software ecosystems, some of which I do not expect to be discovered without experimentation. For example, it can be expected that some information must not leak outside known application boundaries. Considering the idea of the framework is to share information implicitly, how should this be solved? Another challenge is to find a way to ensure that even with groving number of applications, the routing solution scales and individual applications do not conflict with each other. Managing multiple software components answering the same service request may also present a problem.

Although a working software framework prototype has already been developed, my research is still on relatively early stages, so the final course for the research is not yet completely fixed. The work presented in this paper can be understood as the first phase that will provide understanding on the challenges associated with using SERF for intra-application communication. In the current plan, the second phase will concentrate on studying the inter-application part.

In short, the ultimate aim of the research is to gain heightened understanding on the challenges associated with unplanned cooperation among software components. Secondarily, the aim is to provide a feasible and working software framework prototype for further work on the topic. Third, the described workflow will provide a number of smaller prototype applications, each providing insight into the application area in question.

6 Conclusion

In this article I have presented my doctoral thesis topic: SERF, a simple service-based software framework utilizing publish/subscribe in low level settings. The purpose of the framework is to facilitate cooperation between individual applications and software components. I have presented my reasoning why a simpler messaging framework is needed. I have described the current framework prototype and explained the reasons for main design principles. I have explained my plan for continuing the work on developing and evaluating the framework. Finally, I have described my expectations for the result of my research.

References

1. Polojärvi, M.: Application framework for utilizing RFID information Master's thesis, University of Oulu, Oulu, Finland (2008) (in Finnish)
2. Polojärvi, M., Riekki, J.: Experiences in Lightweight Event Relaying Framework Design Proceedings of FutureTech-10, Busan, Korea (2010)
3. Android, Developer Guide: Intents and Intent Filters, `http://developer.android.com/guide/topics/intents/intents-filters.html`
4. Nokia Research Center, NoTA Architecture, `http://www.notaworld.org/nota/architecture`
5. SourceForge, Smart-M3, `http://sourceforge.net/projects/smart-m3/`
6. Thomas Erl, The Service-Orientation Design Paradigm, `http://www.soaprinciples.com/p3.php`
7. Broder, A., Mitzenmacher, M.: Network Application of Bloom Filters: A Survey Internet Mathematics, vol. 1(4), pp. 485–509