Agilo: A Highly Flexible Groupware Framework

Axel Guicking, Peter Tandler, and Paris Avgeriou

Fraunhofer IPSI,
Dolivostrasse 15, D-64293 Darmstadt, Germany
{axel.guicking, peter.tandler, paris.avgeriou}@ipsi.fraunhofer.de

Abstract. Today there exist many frameworks for the development of synchronous groupware applications. Although the domain of these applications is very heterogeneous, existing frameworks provide only limited flexibility to integrate diverse groupware applications in a meaningful way. We identify five variation points that a groupware framework needs to offer in a flexible way in order to facilitate the integration of diverse groupware applications. Based on these variation points, we propose a groupware framework called Agilo that tries to overcome the limited flexibility of existing frameworks by offering multiple realizations of these variation points and providing a modular architecture to simplify the integration of applications and the extensibility and adaptability to different application and integration requirements.

1 Introduction

Today there exist many frameworks to support and to simplify the development of applications for synchronous groupware [1]. While the application domain of these applications covers a diverse range from simply structured and inherently conflict-free applications like chats to conflict-rich shared whiteboards and shared knowledge maps with highly structured data models, the combination of diverse applications from this domain requires the integration on different levels: user interface, application logic, and data model.

The difficulties of combining different applications are caused by their use of different concepts and abstractions, such as different object sharing approaches and distribution architectures. While there are many groupware frameworks that provide certain concepts and several frameworks that offer flexibility in some aspects, yet, there is no framework that offers enough flexibility to combine very heterogeneous groupware applications. In addition, different frameworks often use different domain-specific abstractions, making it hard for application developers to learn a new framework and difficult for them to combine different frameworks [2]. The groupware frameworks we developed in the past [3,4] had a focus on cooperative hypermedia systems, e.g. [5]. Although these frameworks provide excellent support for modeling complex object structures they are too heavy-weight to design applications that don't benefit from using shared objects with transaction-based conflict management and replication support (such as chats or voting tools). However, many groupware systems benefit from the combination of such simple tools with complex ones, making it necessary to combine applications with different requirements.

In this paper we present a new Java-based framework called Agilo that seeks to overcome the shortcomings of existing groupware frameworks with respect to their limited flexibility. Although Agilo supports application integration on all above levels, we concentrate on the two latter levels and describe how it provides the required flexibility by offering multiple realizations of several architectural commonalities of synchronous groupware applications.

The structure of the rest of this paper is as follows: in section 2 we identify several variation points common to synchronous groupware applications and how they are realized in existing groupware frameworks. Section 3 presents how these variation points are realized in the Agilo framework. The final section 4 concludes the paper with a short summary as well as the current status of the framework development and future work.

2 Analysis of Variation Points and Related Work

The diversity of synchronous groupware applications demonstrates that, depending on the specific application, there are many different requirements for the underlying framework. In this section, we identify five *variation points* (also called hot spots [6] or hooks [7]) that represent the aspects of groupware applications which may differ from one to another. The essential requirement for a groupware framework as proposed in this paper is the ability to combine different manifestations of each of the following variation points on the framework level in order to be able to combine different groupware applications (Fig. 1). The identified variation points are a starting point to characterize different types of groupware applications. A more comprehensive analysis whether the identified variation points are sufficient requires further research.

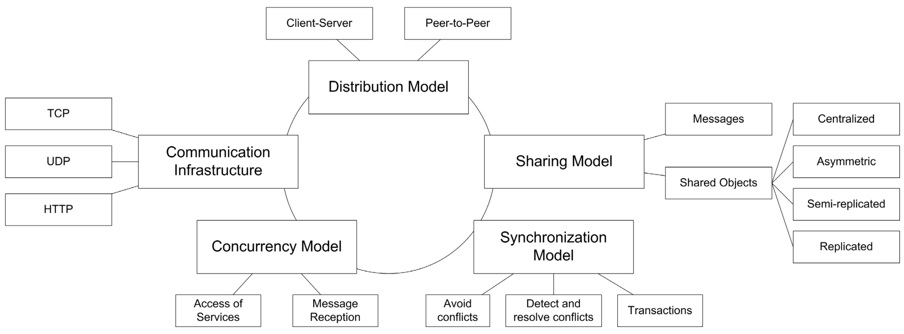

Fig. 1. The variation points and important realizations

Distribution Model. The first aspect that requires variability is the distribution model [1]. The two most common forms are Client-Server and Peer-to-Peer. Most other alternatives can be mapped on a combination of these two approaches [8].

There are many groupware frameworks that support a Client-Server architecture, such as COAST [3] or Rendezvous [9], while others are designed as Peer-to-Peer systems, e.g. GroupKit [10][1] or DreamTeam [11]. Depending on the usage context, each approach has benefits and liabilities. While Client-Server reduces complexity and simplifies consistency issues as well as support for latecomers [1], Peer-to-Peer avoids having the server as bottleneck and single point of failure [8,11]. Additionally, Client-Server is more appropriate when using handheld devices because of their limited resources – the central server then also plays the role of a storage medium. In some circumstances it might even be better to use a hybrid approach where some clients communicate mediated by a server while others form a Peer-to-Peer subgroup [1].

In order to be able to integrate applications that use different distribution models and to adapt the distribution model of an application to domain-specific and environmental constraints, it is essential that groupware frameworks allow flexibility in choosing and adapting the distribution model accordingly.

Communication Infrastructure. The requirements of groupware applications often influence the selection of the communication infrastructure. This includes support for different protocols and different marshalling.

Multiple *communication protocols* must be supported in different application contexts. For clients running on a LAN, a fast protocol such as TCP or UDP is sufficient. If the communication must cross firewalls, it might be necessary to use HTTP or another protocol that firewalls support. Therefore all recent messaging protocols such as SOAP[2] and XMPP[3] are defined independently of the underlying communication protocols.

Similarly, different contexts and applications introduce different requirements for the data exchange format which we call *marshalling*, i.e. the transformation of messages into machine-independent format appropriate for sending through the network. If large amounts of data have to be transmitted, the marshalling should be designed to reduce the size of the data, which may include a binary encoding and compression. On the other hand, if small or heterogeneous devices are communicating, the marshalling must be designed in a way that allows all devices to support it. While SOAP and XMPP support different protocols, they offer an XML-based marshalling only. The COAST framework [3] allows the use of different marshalling for different clients, but uses a single proprietary protocol based on TCP.

Sharing Model. For some applications such as a simple chat application, it is convenient to use messages to inform other clients about application state

[1] Although GroupKit relies on a central server called "Registrar" the communication between the clients is based on a Peer-to-Peer distribution model.

[2] http://www.w3.org/TR/soap/

[3] http://www.ietf.org/rfc/rfc3920.txt

changes. However, applications like a shared knowledge map have a complex object structure that can be implemented far more easily on top of a higher-level abstraction than messages, e.g. shared objects.

While messages are transient objects that carry specific semantics such as a text message in a chat session and that are usually sent only once between two nodes in a system, shared objects are long-living and often persistent objects that are manipulated and updated often by different users. Therefore, the access of shared objects needs to be synchronized in order to avoid data corruption and inconsistencies (see below). Furthermore, the shared objects are distributed in the system using a distribution scheme – typical distribution schemes are central, asymmetric, semi-replicated, and replicated [1].

To free the developer from the burden of implementing concurrency control strategies and distribution schemes as part of the application the framework needs to provide an object-sharing abstraction that includes these two aspects. However, there are two essential requirements: First, the developer must be able to adapt the different aspects of the sharing model in order to optimize the use of available resources such as network traffic and performance. Second, the developer must not be required to use this shared data abstraction at all to avoid potential performance overhead. Besides, for applications where no conflicts can occur (such as chats) or that rely on concepts that do not fit to a shared data abstraction, using shared data is of less value.

Several groupware frameworks directly support sharing of information, such as COAST [3], Rendezvous [9], and DyCE [4]. However, these systems force the developer to use the sharing abstraction. GroupKit [10] allows the combination of both approaches, messages and shared objects.

Concurrency Model. Due to the nature of the domain of synchronous groupware each such application has to deal with concurrency issues. To let application developers concentrate on the application logic, groupware frameworks need to make use of an efficient concurrency behavior. This includes the potentially concurrent access of services by different clients as well as the combined use of asynchronous and synchronous application components without degrading non-functional quality requirements, such as performance, robustness, and scalability. With respect to the sharing model the framework has to correctly resolve the concurrent reception of messages and concurrent manipulation and access of shared objects, respectively.

For example, DreamTeam provides support for interweaving synchronous and asynchronous communication using the Half-Sync/Half-Async pattern [11]. The COAST server uses the Active Object pattern to process incoming messages [3].

Synchronization Model. When concurrent processes access shared resources, synchronization is necessary in order to ensure consistency of data in case of concurrent modification. There are two principal approaches to ensure consistency: *Avoid* conflicts by locking data before modification or to *detect and resolve* conflicts. Common locking mechanisms include mutexes and semaphores. Common conflict resolution mechanisms include transactions and protocols for updating shared data. Both approaches can be implemented in many different ways. Locking is appropriate if, e.g., changes are hard to detect or complicated to resolve. However, locking reduces performance, as the application has to wait for the lock before being able to continue, which affects the usability of interactive systems.

Depending on application requirements, both strategies can be appropriate and therefore groupware frameworks need to provide enough flexibility in this respect. For example, DyCE [4] offers optimistic transactions, whereas COAST [3] additionally offers pessimistic transactions, both with automatic conflict detection and rollback.

3 Framework Design

The design of the Agilo framework directly addresses the variation points described in the previous section. It is based on experiences with groupware frameworks we developed in the past [3,4]. Its flexibility is increased by using design patterns from the domain of distributed and concurrent computing. This leads to an extensible and flexible groupware architecture that allows the integration of heterogeneous groupware applications while giving developers enough freedom in choosing abstractions that fit best to the applications they are building.

Before describing how the different variation points are realized in the Agilo framework, the core concepts of the framework are described next.

The Agilo framework is designed around two key concepts: *Modules* and *Messages*. Modules are software components that are either located on the framework level or on the application level. An Agilo groupware application consists of several modules each running on a node of the system. Messages are application-specific data chunks that are sent between nodes. Incoming messages at a node are processed sequentially and are "forwarded" to one or more application modules which usually send messages to one or more modules running on other nodes as result of processing an incoming message. Providing this message-based communication concept the framework allows the development of groupware applications with a very simple need for communication support such as chats and voting tools, while more sophisticated communication needs can be built easily on top of the message communication (see below).

The framework core is designed to be as small as possible whereas most of the functionality is implemented as separate modules. This approach reveals two advantages: first, the knowledge about the framework required to build applications is kept very small and it can be extended successively as needed. Second, many parts of the framework can be configured independently, leading to a high adaptability and flexibility of the framework.

The remainder of this section elaborates on how exactly the framework provides this flexibility by offering alternative realizations at the different variation points described in the previous section.

Distribution Model. The distribution model of Agilo has been designed to accommodate both the Client-Server and Peer-to-Peer distribution architectures.

In order to be able to establish a Client-Server distribution, the framework consists of three parts: A client-side part, a server-side part, and a common part that is required by both client and server. The Peer-to-Peer distribution is achieved by making each participating node a combined client and server, i.e. by deploying client and server components together in each node. Additionally, both distribution architectures require specific configuration settings in order to adapt the server and client functionality to work in the respective distribution type.

Communication Infrastructure. The Communication Infrastructure of Agilo allows the use of different transport and data protocols. It is realized by following the Client-Dispatcher-Server pattern [12]. The communication between client and server or among peers can be customized on two levels: On the lower level, Agilo supports different transport protocols, such as TCP or HTTP. Protocol-specific implementations accomplish sending and receiving messages while hiding implementation details such as fault-tolerance and native resource handling. The upper level provides different marshalling behavior to support different data-exchange protocols (e.g. SOAP, XMPP). The customizable marshalling behavior especially allows the integration of heterogeneous clients, such as PDAs and smartphones.

Sharing Model. Besides the core concept of “low-level” messages, the Agilo framework offers support for “high-level” shared objects that are implemented on top of the two core concepts of Agilo, messages and modules.

Agilo provides a concrete interface for objects that need to be shared while the distribution scheme is implemented in a separate ObjectManager module. The data itself and its distribution scheme are thus decoupled, allowing the use of different distribution schemes such as centralized, semi-replicated, or replicated shared objects. Application-specific objects that need to be shared have to implement a specific interface in order to be managed by the ObjectManager.

Concurrency Model. The Concurrency Model of Agilo makes provision how multiple concurrent threads can simultaneously work together in the context of the groupware application. Specifically, clients can interweave synchronous and asynchronous messages following the Half-Sync/Half-Async pattern [13].

Another concurrency concern is the processing of incoming messages on the nodes of the system. Messages are received by the node’s ConnectionHandler module following the Reactor pattern [13]. The incoming messages are unmarshalled and enqueued in the node’s MessageHandler module. The messages are dequeued by a single-threaded active object [13], called MessageRouter that is responsible for notification of the node’s application modules. A different implementation of the ConnectionHandler uses the more performant Proactor pattern [13]. Furthermore, instead of the naive single-threaded MessageRouter, a multi-threaded implementation using the Leaders/Followers pattern [13] can be used for module notification if there is no need for a *globally* consistent order of messages.

In case of a Peer-to-Peer distribution model concurrency issues arise because the order of incoming messages is no longer guaranteed to be the same on all peers. The handling of these problems when using a Peer-to-Peer setting is part of the communication infrastructure.

Synchronization Model. The Synchronization Model of Agilo uses different locking mechanisms, such as semaphores and mutexes as well as Java's built-in synchronization mechanisms to enforce controlled access to shared data. Additionally, it allows for detection and resolution of conflicts. The framework supports the use of transactions to combine multiple actions of a client into an atomic action. When a client commits a transaction, a single message containing the manipulations of the affected shared objects is sent to the server where it is processed like any other message. Since the MessageHandler module processes messages sequentially in the order they arrived, the processing of incoming messages uses an implicit transaction management. In the case of a Peer-to-Peer distribution model, the same concurrency issues arise as described in the previous subsection.

4 Conclusions

In this paper we identified the limited flexibility of existing frameworks for synchronous groupware applications as a significant shortcoming in order to combine heterogeneous groupware applications in a reasonable way. This paper focused on the integration of groupware applications on the application logic and data model levels that were partitioned into five different variation points. Underlying frameworks need to support these points of synchronous groupware applications in a flexible and configurable way. Since existing groupware frameworks lack the required flexibility we proposed a new groupware framework called Agilo that seeks to overcome this shortcoming by providing enough flexibility and extensibility with respect to all identified variation points. By providing a very modular architecture that clearly separates different concerns it offers the required flexibility to be applicable for a wide variety of groupware applications.

Since this paper presents work in progress some of the features described above are not yet fully implemented in the Agilo framework. The Client-Server and a rudimentary Peer-to-Peer distribution model, the communication infrastructure, the concurrency model as well as parts of the sharing and synchronization models are already implemented as described in the previous section. However, several essential parts are still missing, such as different distribution schemes of shared objects and a transaction-based synchronization model.

Although the framework is not yet completely implemented, experience derived from its use in a large commercial meeting support system[4] has already proved that the architecture of the framework greatly simplifies the development of synchronous groupware applications. In order to integrate the meeting support system with support for distributed meetings we used an existing chat application framework[5]. In this framework we ported the lower communication

[4] http://www.ipsi.fraunhofer.de/digital-moderation

[5] http://www.ipsi.fraunhofer.de/concertchat

level to Agilo which made it easy to access the generated meeting documents from the chat and provide a tight integration of the two systems.

Besides the implementation of the missing parts mentioned above, the next steps concerning the proposed framework include more case studies, i.e. implementing other diverse groupware applications. Furthermore, the evaluation of the framework concepts and how these support application developers as well as how different combinations of variation point alternatives influence quality requirements such as scalability and performance remain open issues.

References

1. Phillips, W.G.: Architectures for synchronous groupware. Technical Report 1999-425, Queen's University (1999)
2. Lukosch, S., Schümmer, T.: Communicating design knowledge with groupware technology patterns. In: Proc. CRIWG 2004. LNCS, Springer (2004) 223–237
3. Schuckmann, C. et al.: Designing object-oriented synchronous groupware with COAST. In: Proc. CSCW'96, ACM Press (1996) 30–38
4. Tietze, D.: A Framework for Developing Component-based Co-operative Applications. PhD thesis, Darmstadt University of Technology, Germany (2001)
5. Streitz, N.A. et al.: DOLPHIN: Integrated meeting support across local and remote desktop environments and liveboards. In: Proc. CSCW'94, ACM Press (1994) 345–358
6. Schmid, H.A.: Systematic framework design by generalization. Communications of the ACM **40** (1997) 48–51
7. Froehlich, G. et al.: Reusing hooks. In Fayad, M.E. et al., ed.: Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons (1999) 219–236
8. Roth, J.: A taxonomy for synchronous groupware architectures. In: Workshop "Which Architecture for What" of CSCW'00. (2000)
9. Hill, R.D. et al.: The Rendezvous architecture and language for constructing multiuser applications. ACM ToCHI **1** (1994) 81–125
10. Roseman, M., Greenberg, S.: Building real time groupware with GroupKit, a groupware toolkit. ACM ToCHI **3** (1996) 66–106
11. Roth, J.: 'DreamTeam': A platform for synchronous collaborative applications. AI & Society **14** (2000) 98–119
12. Buschmann, F. et al.: Pattern-oriented Software Architecture. A System of Patterns. Volume 1. John Wiley & Sons Ltd (1996)
13. Schmidt, D.C. et al.: Pattern-oriented Software Architecture. Patterns for Concurrent and Distributed Objects. Volume 2. John Wiley & Sons Ltd (2000)