A Framework Designed for Synchronous Groupware Applications in Heterogeneous Environments

Axel Guicking[1] and Thomas Grasse[2]

[1] Fraunhofer IPSI, Dolivostrasse 15, 64293 Darmstadt, Germany
axel.guicking@ipsi.fraunhofer.de
[2] Jeppesen GmbH, Frankfurter Strasse 233, 63263 Neu-Isenburg, Germany
thomas.grasse@jeppesen.com

Abstract. The recent proliferation of using mobile devices in collaborative scenarios increases the need for sophisticated and flexible groupware frameworks for heterogeneous environments. This paper presents the architectural design of Agilo, a groupware framework that has been designed explicitly for synchronous groupware applications involving the use of heterogeneous devices. By respecting device heterogeneity from the ground up, the framework provides an architectural design that is highly flexible along different architectural dimensions on the one hand and simple yet powerful to use on the other hand. Two applications from different application domains based on Agilo are described together with first usage experiences from the developer's point of view.

1 Introduction

During the last decade, the use of mobile devices in daily work scenarios has massively increased. Although mobile devices have found their way into business work settings the application areas still are most often limited to individual services like synchronizing personal calendars, note-taking, and browsing the web. More recently, the research on the integration and use of mobile devices in collaborative settings is constantly growing. It has been pointed out that the use of application frameworks is an adequate way to simplify the design and development of applications in general and groupware in particular [1,2]. Allowing for the increasing trend of mobility in CSCW (Computer-Supported Collaborative Work) scenarios, new groupware application frameworks have been developed or existing groupware frameworks have been extended in order to support mobile devices.

Heterogeneous environments comprising mobile devices exhibit specific characteristics [3,4,5], most notably these are (a) the limitations of processing and battery power, memory and user interface capabilities, (b) unreliable network conditions, and (c) a highly dynamic environment during application runtime including, for example, changing user and device locations).

These characteristics affect all parts typically available in a groupware framework—communication abstractions, framework and application layer, and

Y.A. Dimitriadis et al. (Eds.): CRIWG 2006, LNCS 4154, pp. 203–218, 2006.

user interface support. The belated extension and adaptation of frameworks therefore leads to conceptual as well as implementation-related mismatches between the framework parts addressing classical groupware scenarios and the parts addressing support for mobile devices. Groupware frameworks that have been designed explicitly to support mobile or heterogeneous devices usually focus on specific application domains, such as meeting environments where mobile devices are used as input devices and to share data (e.g. Pebbles [6] and SharedNotes [7]) and collaborative face-to-face learning environments (e.g. ConcertStudeo [8]).

This paper presents the groupware development framework Agilo that was explicitly designed to support heterogeneous devices from the ground up. The consideration of device heterogeneity from the very beginning of the framework design phase has lead to a framework architecture that avoids the above mentioned mismatches while providing support for the different characteristics of heterogeneous environments. By providing a high degree of flexibility along several architectural dimensions the framework is suitable for applications in very different application domains.

The focus of this paper lies on the presentation of the architectural design of the framework and how it meets the characteristics of heterogeneous devices. The framework has been used to build two applications from the domains of public safety organizations and meeting support systems. Besides the description of the applications first usage experiences from the developer's point of view are presented as well.

The remainder of this paper is organized as follows: section 2 motivates the need for a highly flexible groupware framework to include heterogeneous devices. In section 3 groupware frameworks supporting heterogeneous devices are presented and analyzed according to their flexibility. In section 4 the architectural design of the Agilo groupware framework is explicated. Section 5 describes two applications built using the Agilo framework. In section 6 experiences from application development and runtime execution are presented. Section 7 concludes the paper with a short summary and several open issues that need to be addressed in subsequent research.

2 Motivation

In order to provide comprehensive support for application development in general, it has been pointed out that application frameworks need to provide flexibility appropriate to the application domain [1]. For the development of groupware applications, several architectural patterns (or "variation points") have been identified [9,10].

These variation points can be divided into static and dynamic variation points. Variation points addressing static characteristics of groupware architectures are the following: (a) The *Distribution Architecture* addresses the distribution in the collaborative application. Prominent examples for different distribution architectures are Client-Server and Peer-To-Peer distribution models. (b) The *Communication Infrastructure* addresses low-level communication issues such as network

and messaging protocols. (c) The *Sharing Model* specifies how data accessible by different users and components is shared and manipulated, for example by exchanging messages or by manipulating replicated objects.

Variation points addressing dynamic characteristics are the following: (d) The *Concurrency Model* addresses the design how multiple concurrent processes and threads execute in the collaborative application and framework. (e) The *Synchronization Model* specifies the coordination of concurrent access of shared data in order to avoid or resolve conflicting changes. Although the latter two variation points usually address issues that are part of the framework, application developers should be able to easily adapt or even exchange the according framework components according to specific application needs.

It is rather obvious that the tight integration of different realizations of variation points simplifies the development of more complex applications [10]. For example, when considering a meeting scenario where the participants are equipped with notebooks in order to provide input for brainstorming and voting sessions: during the meeting, after one of the voting sessions, the facilitator notices that another brainstorming session should be performed next. He updates the meeting agenda accordingly and changes the configuration of several subsequent voting sessions. While the input of participants is usually entered once and never changed again, the agenda and session configurations need to be synchronously updated in an atomic way at each device. For participant submissions a message-based approach is convenient since they are atomic by themselves and no concurrency conflicts can arise. However, atomic manipulations of multiple data instances necessitate the use of transactions, and, depending on the frequency of concurrent (and maybe conflicting) changes of the data objects, specific concurrency control mechanisms might be necessary as well.

Coming from this example it is only a small extension of the application scenario to include heterogeneous devices which puts other constraints and requirements on the application and, as stated above, on the underlying application framework as well.

3 Related Work

There exists a wide variety of frameworks that provide comprehensive support for the development of groupware applications. However, the support of heterogeneous devices often has been added belatedly to existing frameworks that originally have been designed to support the application development for desktop and PC-based groupware applications, e.g. Pocket DreamTeam [11] or Manifold [12]. During the last few years frameworks have been proposed to support the development of groupware applications using either mobile devices exclusively or using heterogeneous devices. According to the focus of the paper, we focus on the discussion of groupware frameworks for heterogeneous environments.

The DOORS system has been designed for asynchronous collaboration in heterogeneous environments which has been extended to support synchronous collaboration as well [13]. Its object framework provides replicated data objects in

order to allow working on shared data while disconnected (so-called coobjects). DOORS offers flexibility according to the Distribution Model by providing replicated servers and according to the Concurrency and Synchronization Models by encapsulating the according framework functionality and providing different implementations. In order to provide different Sharing Model implementations, the coobject notion needs to be extended. However, these implementations are based on replicated objects which complicates more low-level implementations, such as plain message-passing.

In [12], Marsic presents the Manifold framework, an extension of the DISCIPLE framework [14] to support heterogeneous devices. It uses a data-centric approach for sharing: while data is shared among all collaborators using XML (Extensible Markup Language) it is presented and adapted according to device-specific capabilities using XSL (Extensible Stylesheet Language). Manifold uses a multi-tier architecture by separating concerns in a presentation layer, domain logic, and collaboration functionality. While DISCIPLE already provides support for heterogeneity on the networking level, Manifold makes use of Java Beans[1] in order to provide support for heterogeneous devices on the application and interaction level. Although Manifold provides an extensible architecture and flexibility according to the communication infrastructure, flexibility related to the other variation points is limited.

Pocket DreamTeam [11] is an extension of the Java-based DreamTeam platform [15] in order to support mobile collaboration. DreamTeam is a Peer-to-Peer based platform for synchronous collaborative applications that makes use of so-called "resources" which form the basis for collaborative applications, e.g. shared texts or shared web pages. Each resource can communicate with their corresponding peer resources using synchronous remote method calls. Pocket DreamTeam handles the restrictions of wireless connections by using remote proxies that mediate state changes between peer resources. These proxies are located on stationary parts of the network and therefore provide reliable network connectivity to other peers that may act as proxies themselves. Another extension of DreamTeam addresses flexibility concerning the Sharing Model, called DreamObjects [16]. However, DreamObjects does not support devices with limited capabilities. Furthermore, flexibility according to the Distribution Model as well as most other variation points is limited. In addition, as described in [11], a DreamTeam application has to be ported to C++ for use in Pocket DreamTeam.

QuickStep is a toolkit designed to support data-centered collaborative applications for handheld devices [17] based on record-based shared data (like to-do lists, calendars etc.). In order to avoid conflicts, only the creator of a record is allowed to modify it which in turn allows fast synchronization of replicated objects. However, it does not provide typical groupware services like session management (all users connected to a server implicitly join a session). Furthermore, QuickStep primarily addresses collaboration of co-located users, e.g. in meeting scenarios to synchronize personal calendars. In fact, according to the variation points described above, QuickStep only provides flexibility according

[1] `http://java.sun.com/products/javabeans/`

to the Communication Infrastructure by supporting different communication protocols.

The BEACH environment has been designed to support synchronous collaboration using heterogeneous devices [18]. As an example application for asynchronous brainstormings using limited devices (in this case Palm Pilot V), PalmBeach has been implemented [19]. However, PalmBeach is a separate application that has been implemented from scratch using a proprietary messaging protocol in order to allow for communication with more capable devices running the BEACH platform. The BEACH platform itself has never been designed for application development involving mobile devices with limited capabilities.

4 Framework Design

The Agilo framework combines approaches of so-called "white-box" and "black-box" application frameworks [1]. White-box frameworks support extensibility by providing base classes to be inherited and pre-defined hook methods to be overridden by application developers. Black-box frameworks provide interfaces to plug-in components into the framework by using object composition and delegation. While white-box frameworks usually require application developers to have intimate knowledge of the internal structure of the framework, they provide better support for the developer in order to adapt internal framework functionality than black-box frameworks. Black-box frameworks, on the other hand, are generally easier to use and extend but hide most of the framework functionality. Agilo provides both, template methods and framework base classes to be extended on the one hand as well as interfaces in order to plug-in application components on the other hand. This approach leads to a major benefit over frameworks that strictly follow one of the two approaches: The black-box parts of the framework are fully sufficient to build less complex applications that can be easily accomplished by less-experienced developers. However, the white-box parts allow the fine-tuning of framework-internal structures and behavior by more experienced developers in order to meet application-specific needs and requirements which have not been foreseen during the framework design phase.

The Agilo framework architecture is composed of three tiers: The bottom tier consists of a network abstraction interface and protocol-specific implementations, the middle tier consists of the mandatory framework core and the upper tier consists of application as well as optional framework components (see figure 1).

The upper tier is the framework part application developers usually are faced with. By providing most of the framework functionality as optional components in this tier Agilo becomes much more flexible and customizable compared to frameworks, where all or most of the framework functionality is contained in the middle tier. The upper tier follows a component-based approach which leads to several advantages: (a) components can be easily reused, (b) components can be configured and adapted independently which simplifies testing and increases flexibility, (c) the API (Application Programming Interface) of the core framework is small and compact and therefore easy to learn and memorize which is

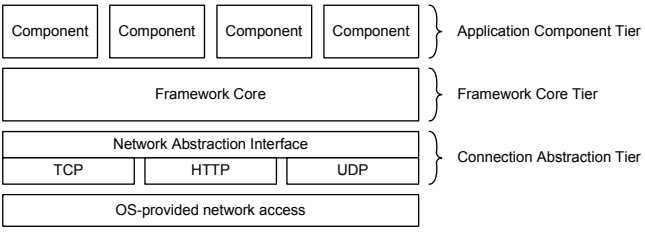

Fig. 1. The Three Tiers of the Agilo Framework Architecture

important especially for less experienced developers, (d) since unnecessary components don't have to be deployed and instantiated, application deployment can be tailored according to specific device capabilities more easily.

The remainder of this chapter details the architectural design of the Agilo framework according to the variation points depicted in section 2.

4.1 Conceptual Model

The Agilo framework is based on three main concepts: *modules*, *messages*, and *connections*. Modules are software components either on framework or application level that are responsible for processing incoming messages (they constitute the upper tier, see figure 1). Messages are application-specific data chunks that are sent between clients and server.[2] The delivery of messages between clients and server is performed by connections that hide low-level implementation details of network protocols (connections are the upper edge of the bottom tier). While the concepts of messages and connections can be directly mapped to the variation points Sharing Model and Communication Infrastructure, respectively, the concept of modules is cross-cutting to the different variation points. The following paragraphs explain the three concepts in more detail.

According to the Sharing Model, Agilo provides different data sharing realizations. As basic communication abstraction Agilo provides messages that are application-specific data chunks sent asynchronously between clients and server. Synchronous messages are realized on top of asynchronous messages that can be used by clients to send a request to the server and block until a response from the server arrives. Although not often required, messages can have an explicit priority in order to be able to process more important messages earlier than other messages. On top of both types of messages and provided as optional components in the upper tier, transactions for atomic sending and processing of multiple messages as well as a generic transaction-based object replication mechanism can be used in application scenarios with a high number of concurrency conflicts and frequent data access and manipulations. The different realizations can be used tightly integrated in a single application.

[2] For the sake of clarity we use the terms Client and Server since a Peer-To-Peer distribution model can be realized on top of the Client-Server distribution model, where each peer acts as both, client and server, at the same time [10].

According to the Communication Infrastructure, Agilo provides a high-level abstraction of network connections (the bottom tier in figure 1) that allows the implementation of applications independent of underlying network and transport characteristics. Typical connection implementations are TCP (Transmission Control Protocol) sockets and—to support nodes secured by a firewall—HTTP (Hypertext Transfer Protocol) connections, where clients constantly poll the server to send and receive accumulated messages. Each connection uses a marshaller that converts messages into a byte sequences and vice versa. By exchanging the marshaller of a connection the messaging protocol can be easily adapted in order to support the integration of third-party clients and devices into Agilo applications or to meet specific security requirements.

An Agilo application usually consists of several modules, each running either on client- or server-side. Modules listen to incoming messages and process them by performing some kind of activity. Which messages a module is interested in is specified by a message filter of the module. The message filter arbitrarily defines a boolean expression to accept or reject incoming messages. This way, a single module can listen to different kinds of messages and different modules can get notified about the same incoming message. Modules are registered at a local *ModuleRegistry* that allows retrieving local module instances using node-wide unique lookup names in order to access application logic of other local modules by direct method calls.

Figure 2 shows the static relationships of the conceptual core of Agilo.

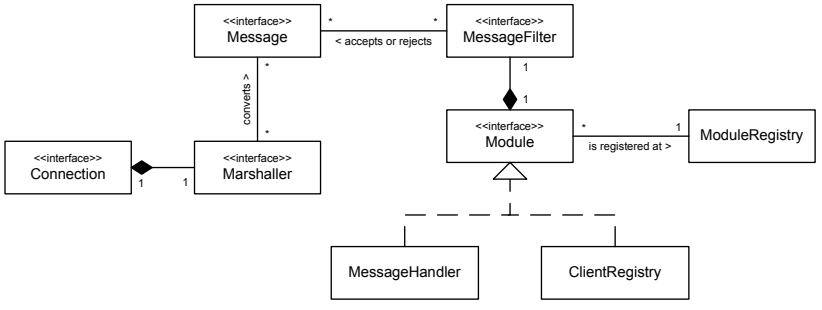

Fig. 2. The Classes and Interfaces Constituting the Agilo Conceptual Model

4.2 Execution Model

During application runtime, several aspects of the Agilo framework core functionality are required in order to provide message delivery and processing. Besides the two variation points addressing dynamic concerns, the Concurrency and Synchronization Model, the typical message flow is of central importance of the Agilo architecture. The following paragraphs describe the two core modules that are related to the dynamic variation points, the Concurrency Model and the

Sharing Model, and that are necessary for the execution of Agilo-based groupware applications before the general message flow in the system is presented.

The server-side *ClientRegistry* module manages the dynamic grouping of clients. By storing a mapping from arbitrary identifiers to a collection of client identifiers, multiple clients can be easily addressed at once to deliver messages. Using the general-purpose ClientRegistry, application-specific session and client management can easily be realized. Since the ClientRegistry is an ordinary module that can be accessed via the ModuleRegistry, the framework does not limit whether the client mappings are entered by clients or directly by the server.

In order to improve network performance the default implementation of the ClientRegistry can be exchanged to support message delivery to multiple clients on network level using, e.g., IP multicast. In combination with a toolkit for reliable multicast such as JGroups[3], the typical drawback of multicast—potential packet loss—can be avoided. However, for scenarios involving widely distributed clients the benefit and performance gain of network-level multicast decreases.

The second core module that needs to be present for processing messages is the *MessageHandler* that supports different realizations of the Concurrency Model. By default it enqueues incoming messages according to their priorities; messages with the same priority are enqueued in FIFO (First In First Out) order. A single active object, the MessageRouter, dequeues messages and forwards them sequentially to the modules that are listening for this message (according to their MessageFilter). In case other message delivery orders are sufficient[4], these can be realized by either configuring the MessageHandler module or, for proprietary concurrent message processing strategies, by replacing it with a proprietary implementation.

Regarding the Synchronization Model, the default implementation of the MessageHandler avoids any conflicts because all messages are processed sequentially, which can be seen as an implicit transaction handling. The use of transactions to bundle multiple messages and process them atomically does not necessitate conflict resolution strategies as well: the messages that are part of the transaction are enqueued one after the other similar to enqueuing single messages—the only difference is that the framework guarantees that no other messages not belonging to the transaction are enqueued in between. Analogously, the execution of transactions that manipulate replicated objects does not require synchronization if manipulations of replicated objects can occur independent of the current object state. Hence, as long as data manipulations cannot fail, no synchronization mechanisms are necessary.

Nevertheless, if manipulations of shared data can fail, synchronization strategies such as locking or automatic conflict resolution are inevitable. To provide support for applications that require this kind of data manipulation behavior, a module providing transaction management based on the Java Transaction API[5] is currently under development.

[3] `http://www.jgroups.org/`

[4] For a survey and comparison of different message orders, see, for example, [20].

[5] `http://java.sun.com/products/jta/`

Figure 3 shows how messages are processed by Agilo. In order to provide a "complete picture," the figure shows a situation where a message is delivered as reaction on an incoming message.

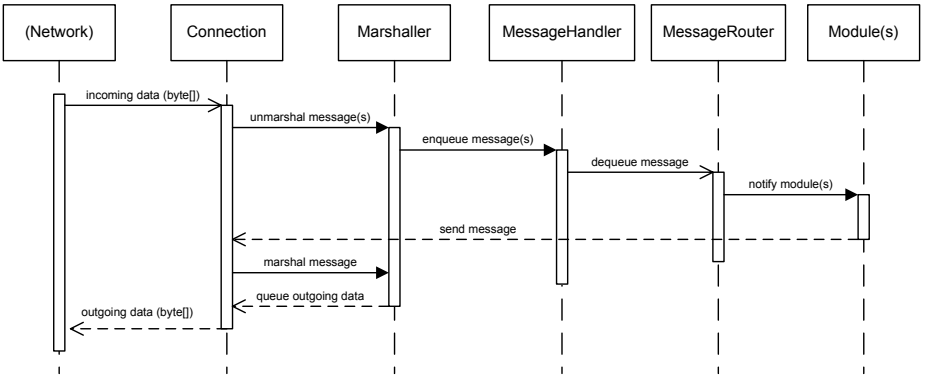

Fig. 3. Message Delivery and Processing in the Agilo Framework

4.3 Immanent Support for Heterogeneous Devices

The Agilo framework addresses the heterogeneity of devices by the following features:

1. The framework provides appropriate device-independent abstractions to free application developers as much as possible from device-specific implementation details.
2. The framework is highly tailorable according to device characteristics and usage purposes.
3. The network abstraction interface provides configurable network protocol-independent reliability support.

The first feature is realized by using Java as framework programming language. For many of the more capable devices on the market today Java Virtual Machines (JVM) based on the Java 2 Micro Edition (J2ME) are available. Devices for that no JVM is available or that do not provide enough resources to execute J2ME-based applications, client applications based on other programming languages can be integrated into Agilo applications by using customized messaging protocols.

The second feature is enabled by the modularity of the framework design. The capabilities of different devices used in a single application scenario often differ. Therefore, the devices are used for specific purposes that best match their individual characteristics. By providing the modularity as integral part of the architectural design of the framework, this massively simplifies application development involving devices with different capabilities:

- Best-matching modules can be chosen by application developers as needed for the purpose of specific devices while modules that are not used by a specific device do not need to be deployed to it. This is a necessary prerequisite for using devices with very limited processing power and memory.
- The runtime performance increases because unnecessary code execution overheads are avoided.
- Since the API of the framework is inherently segmented into the core API and separate module APIs, especially less experienced application developers benefit by not getting overwhelmed by a huge and complex API.

The third feature is part of the network abstraction tier. According to the fact that wireless network connections can be highly unreliable Agilo has to provide a reliable messaging service. In order to provide reliable network connections, the different connection implementations need to be equipped with a guarantee for (a) lossless message transmission, (b) correct message reception, and (c) correct message arrival order without duplicates. These requirements are implemented on framework level instead of completely relying on network protocol characteristics which on the one hand simplifies the connection implementation using other network protocols and increasing code reusability and, on the other hand, enables reliable message exchange independent of the underlying network protocols and marshalling of messages. However, different protocols per se already ensure some of the required reliability issues. For example, TCP provides correct message reception and arrival order, while UDP (User Datagram Protocol) does not provide lossless transmission and correct arrival order.

In order to avoid unnecessary overhead on the framework connection layer, the different connection implementations only make use of the reliability features if necessary. For that, an "optimized ACK" protocol is provided by the framework where lost or corrupt messages are explicitly requested by the receiving from the sending node. In order to be able to use third-party messaging platforms that provide reliable messaging on their own (for example, JGroups or JMS[6]), the reliability features of the framework can be easily switched off.

5 Applications

Based on Agilo, two applications for heterogeneous environments have been implemented: First, an application supporting communication and coordination in emergency missions of public safety organizations, called OPUS. Second, a commercial application for sophisticated large-scale meeting support, called Digital Moderation[7], has been extended to support the use of heterogeneous devices.

5.1 OPUS

The communication during emergency missions as they are performed today by public safety organizations is based on analog trunked radio which leads to several

[6] `http://java.sun.com/products/jms/`
[7] `http://www.ipsi.fraunhofer.de/digital-moderation`

problems [21]: (a) A partner has to follow the whole communication in order to decide which information is dedicated to him. (b) In order to communicate a partner has to interrupt his current work context. (c) In high noise areas the understanding of the communication partner can become difficult and may lead to delays in case of explicit inquiries. (d) Messages that contain a high amount of information probably have to be written down. (e) No private information can be exchanged between two communication partners. (f) Finally, the access to the communication media is not easy as the number of participants increases.

In order to address these problems caused by the trunked radio technique as communication medium, the synchronous groupware OPUS has been proposed in [21]. The requirements for the software were derived from several typical scenarios in missions of public safety organizations. The functional requirements can be divided into the domains task management and resource management. The task management comprises all activities of generation, assignment, and maintenance of tasks. A task is a problem which a resource has to work on. A resource, in turn, is every unit, single man, or equipment that can perform or can be used to perform the work to solve a task. The resource management comprises all activities to control the relationship among resources.

Besides the functional requirements, two non-functional requirements have been identified as well. First, to adequately support the work context the system has to run on handheld devices. This avoids additional weight to be carried by relief units besides their regular equipment. Depending on the user-specific work context, PDA, SmartPhones as well as cell phones need to be supported. Second, during a mission, the device may be not always connected to the network. Thus, interrupted communication links need to be taken into account.

To meet the denoted requirements, the OPUS software architecture has been designed as described in [21]. By applying the patterns "Replicate For Freedom" and "Mediated Updates" [22], the architecture follows a replicated approach, where applications and shared data objects are replicated to client devices. In case of local modifications the client notifies a central server component that propagates the changes to the affected clients. This architecture ensures that a user can still keep on working while the communication link to the server is temporarily interrupted. To support limited devices, the provision of a central server exempts client devices from maintaining lots of communication links.

Figure 4 shows the overall architecture of the OPUS system. The consistency module is responsible for communicating local data changes to the server and for updating local data replicas in case of data update messages received from the server. The data model holds the domain-specific application data which is accessed and manipulated either by the consistency module or locally by the user via the user interface. The task and resource management modules contain the application logic which connects the application data with the user interface.

Details about the implementation of OPUS can be found in [21]. Table 1 shows the realization of the static variation points in the OPUS system. The transaction-based messages are used in order to allow reassigning resources to another supervisor which has been realized by using the Agilo ClientRegistry.

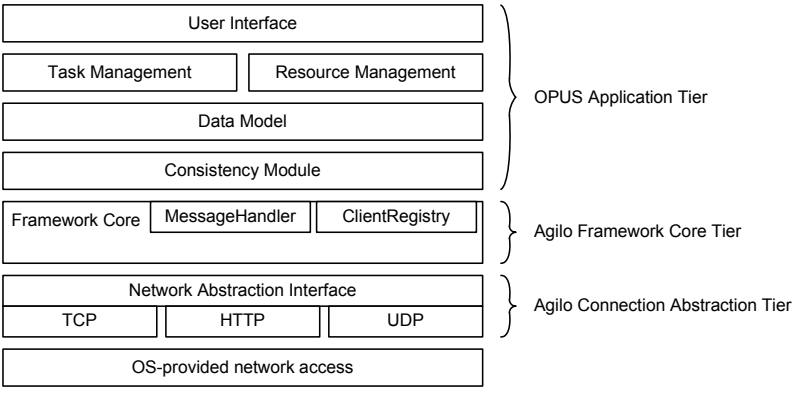

Fig. 4. Top-Level Software Components in the OPUS System

Table 1. Realizations of Static Variation Points in the OPUS Application

Variation Point	**Realization**
Distribution Model	Client-Server
Comm. Infrastructure	Different Network Protocols (TCP, HTTP)
Sharing Model	Asynchronous messages, synchronous messages, transaction-based messages

5.2 Digital Moderation

The Digital Moderation system is a commercial meeting support system for facilitated and co-located meetings providing conceptual as well as technical scalability with respect to the number of meeting participants (meetings with up to several hundreds of users are supported). The main characteristics of the Digital Moderation system are easy adaptability and extensibility to accommodate different facilitation methods, dynamic runtime extensibility to allow changes of the agenda during the meeting, automatic meeting report generation as well as sophisticated facilitation services in order to increase meeting performance. Facilitation methods are realized by providing a meeting tool API, e.g. to provide brainstorming, ranking and voting tools.

Digital Moderation supports different meeting scenarios; besides large-scale meetings, workshop scenarios with about 20 participants are supported that usually are performed by an external facilitator equipped with several WiFi-capable notebooks at a company site (see figure 5). Both scenarios essentially require a very high system reliability. The system itself needs to be robust against hardware and network failures either because no sophisticated failure-tolerating hardware is available or because of financial reasons in case of a large number of meeting participants.

Digital Moderation is implemented using Agilo with a Client-Server Distribution Model which provides better technical scalability for large-scale meetings

compared to a Peer-to-Peer Distribution Model and which simplifies automatic meeting report generation. Network failures are already avoided by the network abstraction tier of the framework while hardware failures, especially in case of server failures, are currently addressed by a generic recovery module based on message logging.

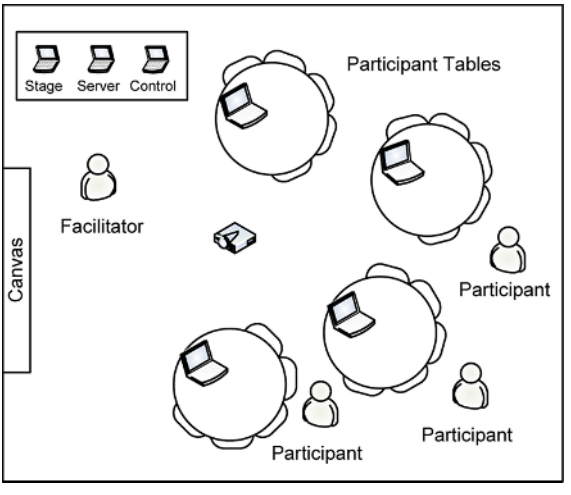

Fig. 5. A Typical Digital Moderation Workshop Setup

Recently, the Digital Moderation meeting scenarios have been extended in order to provide support for using heterogeneous (especially mobile) devices during meetings; the implementation of these extensions are currently ongoing. One of these extensions addresses the technical support during a large-scale meeting: technicians are equipped with PDA during meetings in order to get continuously informed about device characteristics and device-specific connectivity details. Additionally, participants can ask for technical support via the user interface which automatically shows up on the user interface of the technicians' PDA. This way, technical issues can be handled more efficiently and user satisfaction can be improved because of the smooth integration of the technicians and their responsibilities into the meeting execution.

Another extension addresses the use of different devices in a meeting according to specific meeting task characteristics. Depending on the task, devices with different interface capabilities are more appropriate than others. For example, for a meeting of a design team, the use of devices with pen-based input is more applicable e.g. to provide scribbles during a brainstorming session, while other participants use keyboard-based input devices to submit new ideas.

Table 2 shows the static variation points in the Digital Moderation application. While asynchronous messages are used for participant contributions, synchronous messages are mainly used during client startup in order to retrieve

Table 2. Realizations of Static Variation Points in the Digital Moderation System

Variation Point	**Realization**
Distribution Model	Client-Server
Comm. Infrastructure	Different Network Protocols (TCP, HTTP)
Sharing Model	Asynchronous messages, synchronous messages, replicated objects

meeting and tool configuration data. The transaction-based manipulation of replicated objects is used by the facilitator to update meeting data, such as tool configurations and the meeting agenda. Since no concurrent data manipulations can happen (only the facilitator is allowed to update the meeting data), no sophisticated concurrency control and synchronization mechanisms are needed.

6 Experiences Gained

Up to now, the Agilo framework has been used by ten application developers whose expertise ranges from less-experienced Java developers to expert Java developers with comprehensive experience in developing distributed systems. The Digital Moderation application has been used successfully to perform meetings with up to 200 participants involving more than 50 devices.

The Agilo framework core consists of 112 classes for clients and 132 classes for the server which result in framework binaries of around 200 KB and 230 KB, respectively, in size. Optional modules, e.g. the transaction-based generic object-replication without sophisticated synchronization and concurrency-handling, consists of 54 classes resulting in around 50 KB binaries on clients and server.

Compared to Agilo, other groupware frameworks, e.g. COAST [2] and DyCE [23], provide more functionality as part of the non-dividable framework core which leads to two immanent limitations. On the one hand, larger frameworks put more constraints on how to use and extend the framework which massively increases the learning time of application developers. On the other hand, these frameworks require more system resources which limits the applicability for heterogeneous environments.

The experiences gained during the development of the applications sketched in the previous section confirm these conclusions. The small framework core and the modular architecture of the framework lead to a very quick understanding of how to implement applications using Agilo (typically far less than a single day)—even for less-experienced developers.

To our experiences, the modularity and flexibility of Agilo substantially supports the evolution of applications. New functionality can be implemented by introducing new kinds of messages and developing independent modules. This way, new functionality can be easily added without affecting stability and correctness of existing application logic. The flexibility to combine different realizations of variation points in a single application massively simplifies the development of applications combining different complexity levels without overly increasing the overall system complexity.

7 Conclusions and Future Work

This paper presented the architectural design of the groupware framework Agilo that has been explicitly designed to support the development of groupware applications in heterogeneous environments. By providing a highly modular architecture where most of the groupware functionality itself is provided as separate modules, the framework offers a high degree of flexibility, which imposes only few usage constraints on the application development. In fact, applications can be built faster due to increased reusability of software components and faster understanding of the provided framework core concepts. The inherent extensibility of the framework provides support for device-specific development and tailoring for a wide range of devices—from desktop PCs and full-featured notebooks to handheld PDA and devices providing only a very limited set of capabilities like SmartPhones. The experiences gained during the design and development of two different systems (one of them a large commercial meeting support system) based on Agilo have shown that even unexperienced Java developers can comprehend the conceptual framework design very quickly and implement applications within the first one or two days after starting to work with Agilo.

Although the framework has been used to implement two different systems, there are three main areas that require further research. One area addresses the implementation of other applications that especially differ with respect to the dynamic variation points. Since both applications presented in this paper do not put strong requirements on synchronization and concurrency control mechanisms (because of their application domains), the experiences regarding the Synchronization as well as the Concurrency Model are still in an early stage.

Another area is to improve runtime support for heterogeneous devices. In the literature of Ubiquitous Computing it has been emphasized that heterogeneous environments are highly dynamic [3]. While some of the dynamic characteristics are already supported in Agilo (e.g. fault tolerance against intermittent network failures), others are not yet supported in an application-independent way, for example, changing user and device locations and moving stateful applications to other devices. In order to support this kind of context awareness, the approach most often suggested is automatic self-adaptation of the system or framework (e.g. in [3]). For convenient support of application developers the Agilo framework should provide according services, especially to support device and application mobility.

Finally, Agilo needs to be evaluated quantitatively regarding runtime performance, stability and scalability. Albeit there have been several performance and load tests conducted for the Digital Moderation system, concrete statements about throughput, failure frequency and scalability of the framework need to be determined in a more systematic and reproducible way.

References

1. Fayad, M., Schmidt, D.C.: Object-oriented application frameworks. Communications of the ACM **40**(10) (1997) 32–38
2. Schuckmann, C., Kirchner, L., Schmmer, J., Haake, J.M.: Designing object-oriented synchronous groupware with COAST. In: Proc. CSCW '96, ACM Press (1996) 30–38

3. Raatikainen, K.E., Christensen, H.B., Nakajima, T.: Application requirements for middleware for mobile and pervasive systems. ACM SIGMOBILE Mobile Computing and Communications Review **6**(4) (2002) 16–24
4. Roth, J.: Seven challenges for developers of mobile groupware. In: Workshop "Mobile Ad Hoc Collaboration" of CHI '02. (2002)
5. Weiser, M.: The computer for the 21st century. Scientific American (September) (1991) 94–104
6. Myers, B.A.: Using handhelds and PCs together. Communications of the ACM **44**(11) (2001) 34–41
7. Greenberg, S., Boyle, M., LaBerge, J.: PDAs and shared public devices: Making personal information public, and public information personal. Personal Technologies **3**(1) (1999) 54–64
8. Wessner, M., Dawabi, P., Fernndez, A.: Supporting face-to-face learning with handheld devices. In: Proc. CSCL '03, Kluwer Academic Publishers (2003) 487–491
9. Avgeriou, P., Tandler, P.: Architectural patterns for collaborative applications. International Journal of Computer Applications in Technology (IJCAT) **25**(2–3) (2006) 86–101
10. Guicking, A., Tandler, P., Avgeriou, P.: Agilo: A highly flexible groupware framework. In: Proc. CRIWG '05, Springer Verlag (2005) 49–56
11. Roth, J.: The resource framework for mobile applications: Enabling collaboration between mobile users. In: Proc. ICEIS '03. Volume 4. (2003) 87–94
12. Marsic, I.: An architecture for heterogeneous groupware applications. In: Proc. ICSE '01, IEEE (2001) 475–484
13. Preguia, N., Martins, J.L., Domingos, H.J.L., Duarte, S.: Integrating synchronous and asynchronous interactions in groupware applications. In: Proc. CRIWG '05, Springer Verlag (2005) 89–104
14. Marsic, I.: DISCIPLE: a framework for multimodal collaboration in heterogeneous environments. ACM Computing Surveys **31**(2es) (1999) Article No. 4
15. Roth, J.: DreamTeam – A platform for synchronous collaborative applications. AI & Society **14**(1) (2000) 98–119
16. Lukosch, S.: Transparent and Flexible Data Sharing for Synchronous Groupware. PhD thesis, University of Hagen, Germany (2003)
17. Roth, J., Unger, C.: Using handheld devices in synchronous collaborative scenarios. Personal and Ubiquitous Computing **5**(4) (2001) 243–252
18. Tandler, P.: Synchronous Collaboration in Ubiquitous Computing Environments. PhD thesis, Darmstadt University of Technology, Germany (2004)
19. Prante, T., Magerkurth, C., Streitz, N.: Developing CSCW tools for idea finding: empirical results and implications for design. In: Proc. CSCW '02, ACM Press (2002) 106–115
20. ter Hofte, H.: Working Apart Together. Foundations for Component Groupware. PhD thesis, Telematica Instituut, Enschede, NL (1998)
21. Grasse, T.: Eine Systemarchitektur zur effizienten Steuerung von mobilen Einsatzkrften [in german]. Master's thesis, FernUniversitt Hagen, Germany (2005)
22. Lukosch, S., Schmmer, T.: Patterns for managing shared objects in groupware systems. In: Proc. EuroPLoP '04. (2004) 333–378
23. Tietze, D.: A Framework for Developing Component-based Cooperative Applications. PhD thesis, Darmstadt University of Technology, Germany (2001)