SelectiveDES: A Distributed Event Service Add-On for Invocation-Based Middleware supporting Selective Multi-Channel Communication and Notification Delivery

Werner Kurschl, Stefan Mitsch, Rene Prokop
Upper Austrian University of Applied Sciences - Research and Development Competence Center
Hauptstraße 117, A-4232 Hagenberg, Austria

Abstract

Mobile enterprise applications typically access data from the enterprise's various applications to support collaborative working processes. Allowing the mobile application to access these data online only would be a major hindrance for mobile workers that cannot assume a constantly available network connection. This problem can be handled by middleware systems, which provide a way to prefetch data on the mobile device. But changes and events on the central data cannot be foreseen; moreover, they cannot be delivered to disconnected mobile clients. We introduce a distributed event service named SelectiveDES for managing and delivering events to mobile clients in unpredictable network environments. SelectiveDES is designed as an add-on to common invocation-based middleware systems; it is based on the publish-subscribe paradigm and supports selective multi-channel communication and notification delivery.

1. Introduction

Mobile enterprise applications in industrial environments often support collaborative working processes. The mobile workers need to stay in contact with a central office to report their current status via a wireless network; industrial environments typically operate different networks-e.g., IEEE 802.11b/g with high bandwidth and low costs but suboptimal coverage, and GPRS with low bandwidth and high costs in favour of high coverage. To keep costs low, data is usually transmitted over networks free of cost. Consequently, mobile workers need to work also in areas with no network coverage. Allowing the mobile application to access data online only would be a major hindrance in such a scenario. Thus, the mobile workers must also have access to enterprise data when being offline. Data needs to be replicated to the mobile device—within the constraints of available local memory—for offline access; changes and updates to these data need to be tracked and synchronized with the enterprise applications when a network connection can be reestablished.

This problem can be handled by middleware systems, which provide a way to pre-fetch data on the mobile device and synchronize local changes with changes on a central server. But changes and events on the central server's data cannot be foreseen; often, data changes are vital to mobile workers. Hence, the server needs to communicate those data changes actively to its clients; it cannot wait until the client initiates synchronization.

To further illustrate these requirements we shortly describe the environment of MOSES (see [6]), a mobile enterprise application for work clearance management in industrial environments. MOSES allows maintenance workers to process the items on a shared to-do-list. These items represent a workflow; one worker has to process certain items, before another worker can begin working on other items (e.g., the electrician needs to turn off power, before the mechanic can start manipulating the conveyor belt). A control centre observes the work flow and adds new tasks to the to-do-list as needed. More importantly, the control centre can identify potentially dangerous areas in the plant and has to notify workers from their existence at all cost to avoid accidents. In the meantime, office workers might work on optional data further describing items on the to-do-list. These changes are often of minor importance, as they typically do not affect the working process.

In the course of the project MOSES we developed an invocation-based middleware, which especially suits unpredictable network environments in connection with mobile clients named SmartDOTS.

We separate a distributed event service, called SelectiveDES, from the common middleware functionality. SelectiveDES manages notifications and selects appropriate communication channels for each notification. A distributed event service allows objects at different locations

to be notified of events taking place at another object (see [3]). We use the publish-subscribe paradigm, in which an object publishes the type of events that are available for observation by others. Objects, that want to be notified of events, subscribe to the types of events that are of interest to them. When the publishing object experiences an event, notifications about the event are sent to all subscribers that expressed interest in that type of event.

Mobile enterprise applications in industrial environments often need special notification delivery semantics that differ from those used in common distributed event services. Notifications need to be sent reliably, but the distributed event service cannot rely on the fact that all clients happen to be online when the notification is sent. Depending on a notification's priority, it needs to be sent immediately—requiring the service to switch to a different network—regardless of cost (vital information), or it can be queued in an event history until a network connection to the client free of cost can be established (minor information). We call this mechanism *selective multi-channel communication and notification delivery*. Queued notifications need to be managed in terms of their validity (notifications have to expire) and consistency. Subsequent notifications might render a previous notification meaningless or they can override it.

Shortly summarized, we identified the following requirements and challenges for our distributed event service that are also typical for other mobile enterprise applications in industrial environments. *(i)* varying network environments (often multiple networks, like WLAN, UMTS, or GPRS) with sporadic connectivity and/or low bandwidth, *(ii)* notification of data changes to keep replicated data in sync, *(iii)* information can be of different importance (reaching from minor to vital), and *(iv)* information can be rendered meaningless or might be overridden.

2. Related Work

Fiege et al. describe in [4] a publish-subscribe middleware called REBECA for spontaneous pervasive applications. Their main contribution is the consideration of device mobility. REBECA consists of a network of brokers that route notifications to the subscribed clients. A mobile client can roam between the scopes of different brokers. Additionally, REBECA defines an entity called virtual client that collects notifications on behalf of the real client at a broker. When the real client emerges the broker, the virtual client hands the collected notifications to the real client. However, their work stops at defining algorithms for efficiently routing notifications through brokers and collecting notifications. They do not describe algorithms for managing events in the queue (time- or semantic-based expiration), nor do they provide prioritization and alternative networks for sending important notifications.

Hermes (see [8]) is a distributed event-based middleware architecture that follows a type- and attribute-based publish-subscribe model. It introduces an event-type that supports features commonly known from object-oriented programming languages (like type hierarchies and supertype subscription) to better support features provided in common middleware. Hermes is designed as an event-based middleware that completely substitutes common invocation-based middleware. In contrast, SelectiveDES is designed as an add-on to existing invocation-based middleware, though some of Hermes' concepts (like type hierarchies) can also be found in our work.

SIENA (see [2]) is an event notification service that aims to maximize expressiveness and scalability. Expressiveness refers to the notification service's ability to filter events and to use these data for optimizing notification delivery. SIENA optimizes notification selection and notification delivery to maximize scalability. Therefore, it proposes a distributed architecture for the notification service itself.

Meier and Cahill (in [7]) describe STEAM, which is an event-based middleware for wireless ad-hoc networks. Their research focus is on event filter types to address the dynamic aspect of the network topology. In their application scenario, a large number of entities can dynamically join and leave the network.

Yaco (see [1]) is a framework for supporting mobile collaborative work. It focuses on user-centric services like messaging and user discovery, as well as on file-centric services like file transfer and search of artifacts. Yaco relies on a constant network environment and ignores issues related to temporal decomposition of collaborative work.

Podnar and Lovrek in [9] propose notification persistency for supporting mobility. They substitute proxy subscribers (typically used for collecting notifications while the real subscriber is unreachable) with persistent notifications and an expiration mechanism. SelectiveDES does not follow this approach; instead, it tries to deliver notifications by having the proxy select an appropriate communication channel. Thus, proxies have to queue less notifications than in ordinary publish-subscribe systems.

3. Architecture

3.1. Integration

This section describes the integration of the distributed event service into client-server applications based on a middleware which provides offline access to the business data. Fig. 1 shows a mobile application based on the middleware SmartDOTS, providing access to the business data model even in unpredictable network environments.

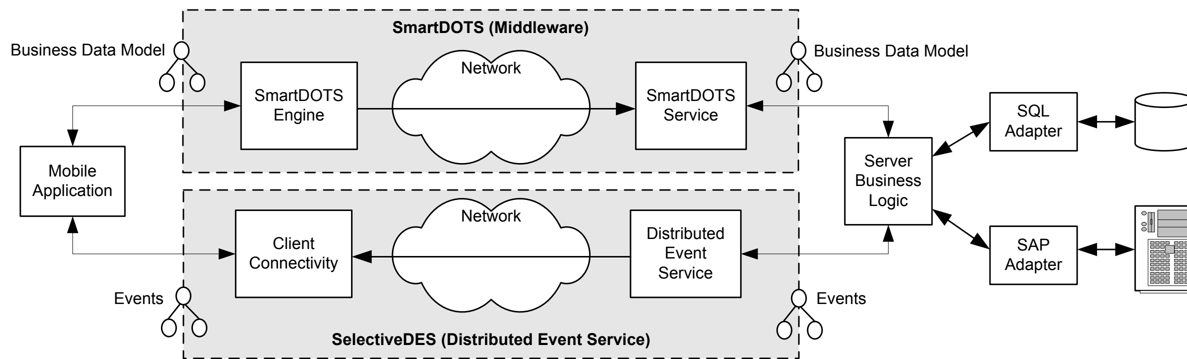

Figure 1. The distributed event service SelectiveDES collocated to SmartDOTS.

This middleware offers replication, offline data management and synchronization to the mobile client. SmartDOTS is—like it is usual for this kind of middleware—invocation-based, which means that the client application is the active part, while the server behalves passively. Especially in service-oriented architectures the server has no possibility to establish a connection to its clients.

As discussed before, there are scenarios in which the clients must be notified. As depicted in Fig. 1, SelectiveDES is deployed side by side to the middleware and uses its own communication channel. A distributed notification mechanism demands for the server actively transmitting events. Therefore, the notification mechanism is based on .NET Remoting. The distributed event service is not tied to the SmartDOTS middleware, it can be used with any other middleware or even stand-alone. But typically it is combined with SmartDOTS because of the scope to similar environments; it can be seen as server-side add-on to SmartDOTS.

3.2. Registration and Management of Clients

SelectiveDES offers two peers: the interface *IClientsReception* for registration of the clients and the interface *IServersReception* by which the server application is able to notify its clients about events on the server. The second interface is described in Sect. 3.3 in detail. Both interfaces are implemented by the class *ClientManager* which is the core of the notification mechanism (see Fig. 2). The ClientManager provides a set of Protocols that can be used for transmitting the notifications. The protocols define a cost model containing the arising expenses and the available bandwidth. The set of supported protocols can be extended by adding a new pair of Protocol and ClientAddress classes to the framework. The gray shaded elements in Fig. 2 (et seq.) depend on the deployment of SelectiveDES and can be added to the framework by defining them in its configuration.

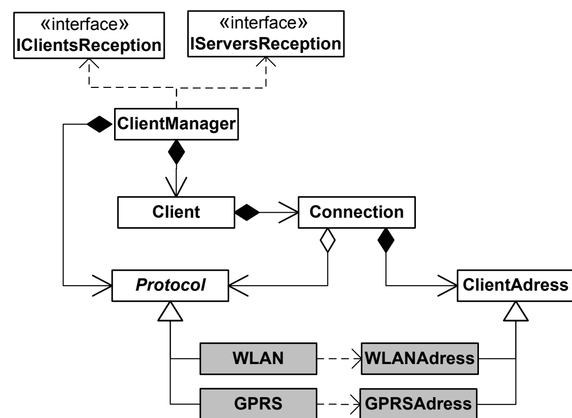

Figure 2. Clients and their supported connections.

Each client that wants to be notified about events on the server must register itself via the interface IClientsReception. By that, the ClientManager receives a remote reference to the client. Depending on the network condition and the priority of events, it might be necessary to use alternative communication protocols (like WLAN, GPRS, etc.) for distributing the events at a later point of time. Thus, the address information can not be implicitly determined from the connection during the registration process. Instead, the client itself has to provide explicit information about which types of protocols are supported and how it can be addressed over these protocols.

After registering the clients can subscribe for different event types it is interested in. By that the communication overhead and costs can be reduced, especially when working with mobile devices with limited resources (like smart phones) in demanding network environments.

The ClientManager manages the registered clients and

their communication options. For each client a time-out is set to remove permanently disconnected clients from the system. The time-out is reset by "keep-alive" messages generated inside the distributed notification mechanism after each successfully transmitted notification. Additionally, "keep-alive" messages can also be provided over the interface IServersReception; thus, the server implementation of the distributed application can also reset the time-out when the client invokes the server.

3.3. Event Management

The distributed notification mechanism supports various events, which must be defined in its configuration. Each event definition contains an event type, a priority, time to live and a cost limit, which its transmission must not exceed.

The events that should be distributed can be triggered over the interface IServersReception. The interface does not accept event objects; the kind of event is defined by delivering its event type. The ClientManager retrieves the corresponding Event object from the *EventFactory* (Factory pattern, see [5]). To keep the number of instantiated Event objects small, the EventFactory holds an *EventPool*, which contains exactly one instance per event type. By that the number of Event objects remains constant independently from the number of clients and triggered events.

The ClientManager passes the Events to the *Clients*, which subscribed for this event type. The Events are not transmitted immediately, but managed in a *NotificationQueue* until it is their turn to be sent. Fig. 3 shows the relationship between the Clients and the triggered Events. The NotificationQueue holds additional information (like a time-out) to the shared Events in *Notifications*. Notifications are small-footprint objects; they represent a single Event at one single Client.

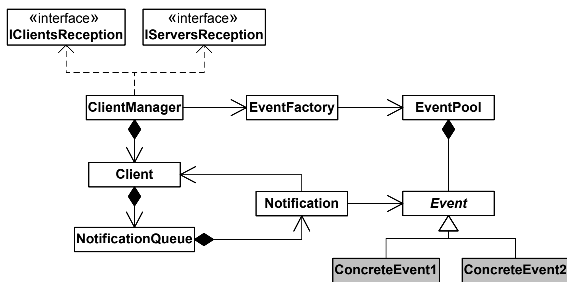

Figure 3. Buffering and queuing of events.

Based on these values, send order and transmission protocols will be chosen later. When a new Event is added to a Client, it must be evaluated individually. The evaluation checks if the Event has influence on already queued events. For example, it may be useless to send an event "list changed" twice, because the client has to update the whole list anyway. Instead, we can increase the already queued event's priority when multiple events of the same event type are queued.

3.4. Transmission of Events

The transmission of the Notifications is a separate, continuous process which is handled by the *EventDispatcher*. The framework contains a single instance of the EventDispatcher, which is not related to single Clients. It handles all queued Notifications globally. That ensures that the high-priority Notifications are sent first, regardless which Client they belong to or how many high-priority Notifications are queued at a single Client.

The EventDispatcher continuously sorts queued Notifications of all Clients according to their priority, queue time and failed transmission attempts. Fig. 4 shows the involved components.

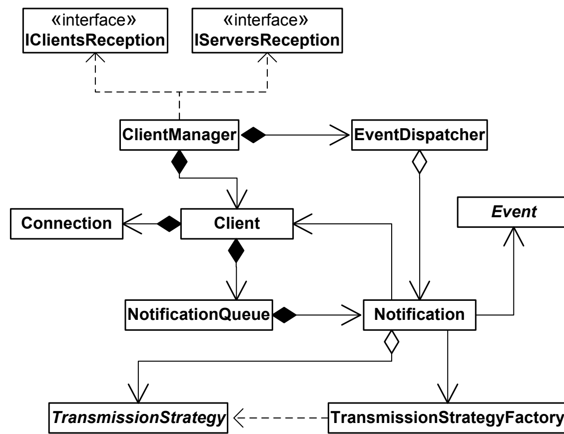

Figure 4. Transmission of events.

The transmission of single Notifications is triggered by the EventDispatcher and performed by a *TransmissionStrategy*. The TransmissionStrategy defines the algorithm of establishing the best connection for a single transmission attempt, which can contain multiple connection establishments over various protocols. The concrete TransmissionStrategy depends on the event type and is created using the TransmissionStrategyFactory. The concrete strategy is determined just in time because most Notifications will expire or will be suspended by subsequent events before reaching this point of process. The TransmissionStrategyFactory works analogous to the EventFactory with its EventPool described before.

The concrete TransmissionStrategy is responsible for sending the Notification to the client in the cheapest and fastest possible way. A possible implementation of the strat-

egy could be to start trying to send using the "cheapest" connection and continue with the next "expensive" connection if the cheaper one is not available. The strategy pattern (see [5]) is used here, because depending on the event type different approaches of finding the best connection can be used.

The TransmissionStrategy does not deal with Protocols directly, but with categories of protocols to make the algorithm simpler and leave the system flexible. By that the usage of SelectiveDES becomes easier too. In each deployment different event types and protocols will be used and must be adapted. This can be done in the configuration, while adapting the deep-inside algorithm like the TransmissionStrategy is much harder and will be done very rarely. But assigning individual Protocols to *ProtocolCategories* is a simple configuration process. We introduced categories of Protocols, on which the TransmissionStrategies are based on to decouple the transmission algorithm and the concrete protocols. Fig. 5 shows an extension of Fig. 4 in which the TransmissionStrategy is not directly linked to the protocol. To illustrate the configurability of the system, the dynamically exchangeable elements are gray shaded.

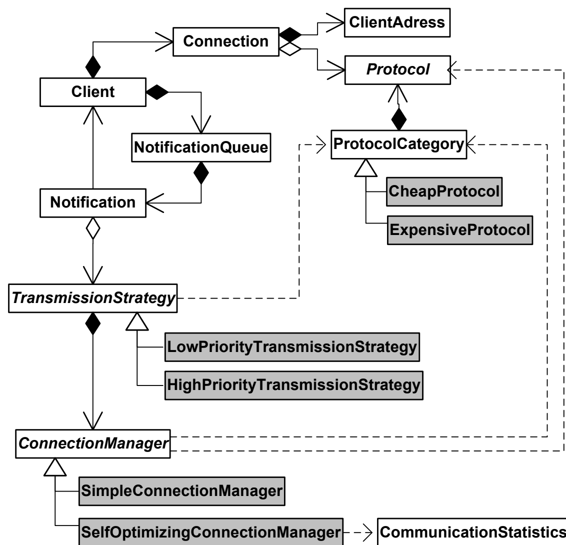

Figure 5. Decoupling transmission strategy and concrete protocols.

By having the TransmissionStrategy based on ProtocolCategories, an additional step becomes necessary to retrieve the concrete Protocol, which is supported by the Client. When the TransmissionStrategy decides to use a specific category of protocols (e.g. *CheapProtocol*) for the Notification currently in process, it uses the *ConnectionManager* to retrieve the concrete protocol, which suits the demanded category and is supported by the Client. It is not sufficient if the ConnectionManager evaluates just the assignment of concrete protocols to categories. It must examine which protocols are supported by the Client, to which the Notification should be sent. Nevertheless the decision which protocol to deliver might not be distinct, because the Client might have registered a LAN and a WLAN connection which are both "cheap" connections.

This is one major point for performance enhancements: While currently the *SimpleConnectionManager* (which bases on the mediator pattern, see [5]) is in use, a statistic-based *SelfOptimizingConnectionManager* could reduce the number of transmission attempts dramatically. The concept is to monitor the transmission attempts per client and build up a *CommunicationStatistics*, which contains the data about the reachability behavior of single clients (i.e., which protocol was used lately). By that the choice between protocols within the same category could be based on statistic presumption and would be self optimizing.

If the transmission succeeds, the Notification is deleted and an internal "keep alive" message is sent to the ClientManager to prevent the expiration of the Client. If the transmission fails, it is assumed that the client is offline. Therefore, the Notification is left in the NotificationQueue and the queue time is modified to ensure, that the Notification is realigned correctly by the EventDispatcher and sent at a later point of time.

4. Conclusion and Further Work

We presented a distributed event service named SelectiveDES that is designed as an add-on to invocation-based middleware. SelectiveDES introduces the following main concepts to the field of event services. Communication is no longer bound to a single network; instead, events determine the type of networks that can be used. SelectiveDES provides strategies to reliably transmit notifications within the constraints defined by events and also minimizes the costs for transmitting notifications.

Further work will comprise the evaluation of different transmission strategies and connection manager implementations. The analysis of a client's communication behavior promises interesting optimization possibilities in our system.

References

[1] M. Caporuscio and P. Inverardi. Yet Another Framework for Supporting Mobile and Collaborative Work. In *Proceedings of 12th International Workshop on Enabling Technologies, Infrastructure for Collaborative Enterprise (WETICE 2003)*, pages 81–86, Linz, Austria, 2003. IEEE.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. 19(3):332–383, 2001.

[3] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems—Concepts and Design*. Addison-Wesley, Boston, 3 edition, 2001.

[4] L. Fiege, A. Zeidler, F. C. Gärtner, and S. B. Handurukande. Dealing with Uncertainty in Mobile Publish/Subscribe Middleware. In *Proceedings of 1st International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC03)*, pages 60–67. ACM Press, 2003.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, 1995.

[6] W. Kurschl, S. Schmid, and C. Domscha. MOSES - A Mobile Safety System for Work Clearance Processes. In *Proceedings of the 4th International Conference on Mobile Business*, pages 166–172, Sydney, Australia, 2005. IEEE.

[7] R. Meier and V. Cahill. STEAM: Event-Based Middleware for Wireless Ad Hoc Networks. In *Proceedings of 1st International Workshop on Distributed Event-Based Systems (DEBS02)*, Vienna, Austria, 2002. IEEE.

[8] P. R. Pietzuch and J. M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of 1st International Workshop on Distributed Event-Based Systems (DEBS02)*, Vienna, Austria, 2002. IEEE.

[9] I. Podnar and I. Lovrek. Supporting Mobility with Persistent Notifications in Publish/Subscribe Systems. In *Proceedings of 3rd International Workshop on Distributed Event-Based Systems (DEBS04)*, pages 80–85, Edinburgh, UK, 2004. IEEE.