Gossip: An Awareness Engine for Increasing Product Awareness in Distributed Development Projects

Babak A. Farshchian

Dept. of Computer and Information Science
Norwegian University of Science and Technology, Trondheim
http://www.idi.ntnu.no/~baf

Abstract. More and more product development projects involve geographically distributed groups of developers. One problem in such groups is the long term lack of awareness of the activities in remote sites. In this paper we discuss the importance of awareness in distributed product development projects. We argue that generic services are needed in development environments for providing continuous awareness of remote sites. We introduce a product awareness model that puts focus on a shared composite product and the propagation of awareness in it. We describe the design and implementation of this awareness model in form of an awareness engine called Gossip.

1 Introduction

Information systems development is an area of intensive human collaboration [BJ75]. A normal practice for managing collaboration in large scale IS development projects has been to divide the system into parts and have the parts be developed separately by groups of developers, in this way reducing the amount of ad hoc communication and dependencies [Par72,Con68]. However, many recent case studies of project groups have revealed that this is not an easy task. No matter how rational the division of the system into parts is, each group will still need a large amount of information about what is happening within and across the groups in order to coordinate its work [HG99,Gri98,KS95]. Access to this ad hoc information becomes particularly problematic when the developers are geographically distributed. The effect of geographical distances on long term collaboration has been well documented in the literature. Herbsleb and Grinter [HG99], Kraut and Streeter [KS95], and Krasner et al. [KCI87] document the occurrences of communication breakdowns when there are organizational or geographical barriers among the group members.

In a study of researchers from 70 research labs, Kraut and Egido [KE88] found that there was considerably higher collaboration frequency among researchers having offices in the same corridor than among those who did not. The reason, according to the authors, is that physical proximity increases the *frequency* and the *quality* of communication, and decreases the *cost of initiating* communication. As members of co–located groups, developers have the advantage of constantly being *aware* of the status of the product being developed by simply using their social abilities, such as "looking over the shoulder" of each other, being involved in chance encounters in the corridors, and having a much greater opportunity for "keeping in touch" with each other. In distributed cooperation, both the amount and the quality of this information decreases. In addition, providing and consuming this *awareness information* becomes an explicit burden on the co–workers simply because most of the natural social channels of communication are eliminated.

In the recent years there has been a large amount of research conducted with the aim of developing tools to support distributed development groups in performing specific and recurring types of tasks. Tools are developed for supporting collaborative modeling [DLN97], JAD sessions [CGN95], programming [HW98], and more. In addition to these specific tools and environments, we believe there is a need for more generic tools for simulating long–term physical proximity of the distributed project members, making it easy and less costly for the developers to initiate collaboration when they need it. In this way strict geographical divisions can be relaxed, and the collaboration can proceed in a more natural and flexible way. One essential step in doing this is to develop support systems that increase the amount of long term awareness provided to project members about the activities of remote co–workers.

As a part of our research on product development environments, we have been developing a framework for supporting collaboration in distributed product development projects. The framework consists of three parts (Fig. 1). An underlying *product layer* is in charge of providing awareness of a distributed virtual product consisting of parts. A *cluster layer* implements mechanisms for grouping product parts into collaboration aware clusters, such as diagrams. An *application layer* assists the integration and use of various development tools.

In this paper we describe *Gossip*. Gossip implements the product layer, and is in charge of keeping all the other parts of the framework constantly aware of user activities involving different parts of the product, in this way providing a shared collaboration space for distributed development groups. The structure of the paper is as follows: In Sect. 2 we will look at different approaches to awareness support. In Sect. 3 we introduce our product awareness model. Section 4 describes Gossip, which implements this awareness model in form of an awareness engine. Section 5 provides a discussion and directions for future work.

2 Approaches to Awareness Support

Awareness is the information that human beings exchange with their environment in order to coordinate their work with others. Providing computer–based mechanisms for supporting exchange of awareness information has shown to be of central importance to the design of collaboration support systems [GG98]. For the purpose of this paper, we will organize the research in awareness support along two dimensions (Fig. 2). The first dimension focuses on the *quantity* of the exchanged awareness information compared to the natural co–located situations.

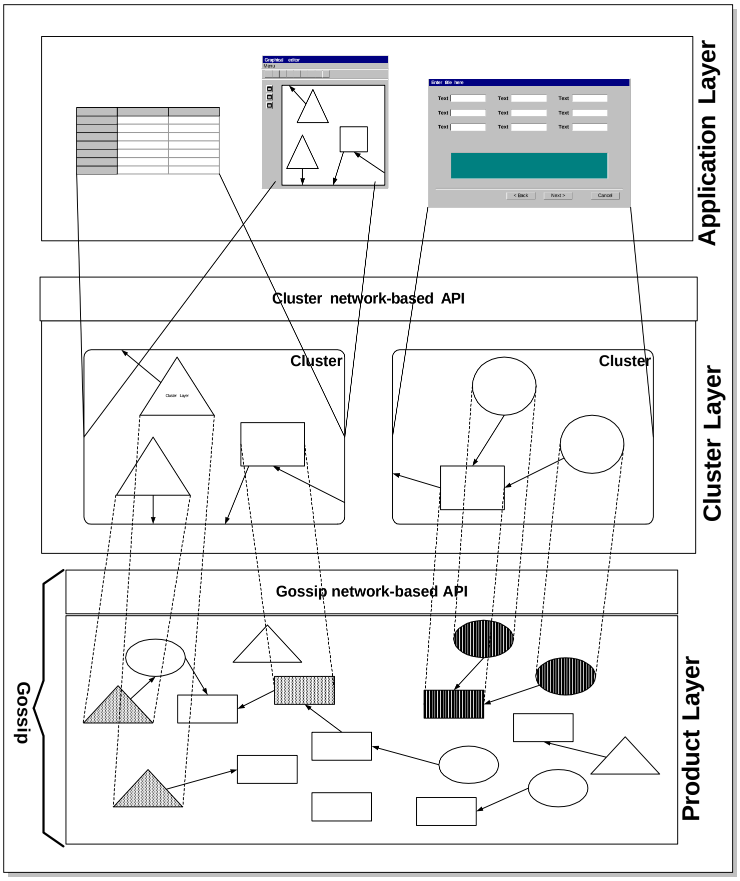

Fig. 1. A framework for supporting product awareness in distributed product development environments.

The second dimension is the level of *organizational awareness* provided by the awareness mechanism. By organizational awareness we mean awareness of the overall organization of the work. High quantities of awareness information are normally important in order to increase the "naturalness" of the collaboration, while access to proper organizational awareness facilitates collaboration in large groups.

The quantity of transmitted awareness information is largest in co–located situations, where all the available social channels are used to their full capacity. *Media spaces* are probably the closest imitations of co–located situations regarding the quantity of awareness information. A typical media space consists of permanent video and audio connections between geographically distributed sites. In addition to the large amount of awareness information, the permanence of the

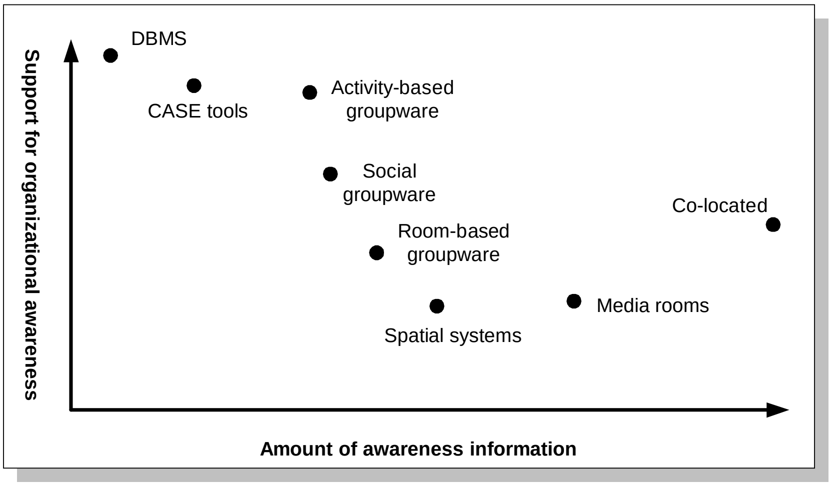

Fig. 2. A comparison of different awareness models.

connections reduces the cost of initiating collaboration, contributing further to the creation of a common social space in the long run [BHI93].

Smaller amounts of awareness information are typically transmitted through software based collaboration support systems. This is both because of the difficulty of collecting this information from the group's natural context, but also because of the difficulty of visualizing this information in a proper way. A challenging part of implementing awareness support is thus to find the proper *awareness model* for the domain to be supported. This model will decide what awareness information will be delivered to whom, in this way both making better use of the limited information bandwidth, and ensuring that those who need the awareness information will get it.

Several awareness models have been proposed in the literature. Normally, awareness models have a notion of *shared space*, *presence* and *focus*. Shared space is where collaboration objects have to reside in order to be able to produce and consume awareness information. An object makes its presence known to the shared space by registering itself with a set of parameters. Once present in the shared space, the objects commit to update their presence information regularly. Objects can in addition *focus on* other objects by providing to the shared space a set of parameters that defines their focus. The focus can also be updated regularly. In the *spatial model* of Benford and Fahlèn [BF93], for instance, each object declares its presence in the shared space by its coordinates according to some spatial measure (e.g. geographical location). By changing its coordinates, an object can "move" in the shared space. The focus of each object is defined in form of an "aura"; each object will be able to receive awareness about other objects that are inside its aura, e.g. are close to it. Moving in the space will also move the object's aura.

The spatial model is mainly used in virtual reality settings. A similar awareness model is the *room–based model*, where awareness of the activities in a virtual room is propagated to all the inhabitants of the room, while the “walls” limit the propagation of awareness information to the outside. This model is used in many shared workspace applications, notably TeamRooms [RG96]. A third model is the *activity–based model*. Here, awareness information delivery is not based on spatial relations, but on the activities the user is currently involved in [FPBP95]. *Social awareness models*, on the other hand, put more focus on human participants and interactions among these. One such model is the *locale* model used in Orbit [MKF$^+$97]. The locale model provides awareness based on “social worlds,” e.g. centers of social activities.

Awareness can be connected to the immediate surroundings of a user, or to a broader scope of happenings in an organizational context (the organizational awareness axis in Fig. 2). The activity–based model used in GroupDesk [FPBP95] provides a good example of high organizational support. GroupDesk is used in conjunction with a workflow application, where a user involved in a step in a workflow is made aware of the activities in other related steps. Spatial and room–based awareness models are more limited in the organizational support dimension since they strictly focus on the physical space surrounding a user. Social models support well the informal organizational structure, which may or may not mirror the formal structure.

Though CASE and similar product–based tools provide access to an organizational context in form of a large shared product, the degree of awareness information exchange is low [VS95]. Central repositories used in product–centered development tools provide a first technological step for supporting collaboration, but more is needed. These repositories are normally developed in form of time–sharing systems. In particular, they are based on the assumption that conflicts among developers should be delayed as long as possible [JMR92], with the consequence of isolating developers from each other. Despite this attempt to isolate, Vessey and Sravanapudi [VS95] found that having access to a shared product and being able to observe the changes done to it by others had an implicit coordinative role in CASE tools. In the next section we will introduce a new awareness model that tries to address this problem by supporting exchange of explicit product awareness among developers.

3 A Product Awareness Model for Distributed Product Development

In this paper we introduce an awareness model that uses the product, in particular its structure, as a basis for providing awareness to the developers in a distributed project. Our assumption, as discussed in the introduction, is that a developer needs to have constant awareness of the activities related to a shared product. These activities are first and foremost related to the parts of the product that a developer is directly working on, but a developer may also need to be informed about the activities related to other parts of the product that are some-how related to "his" parts. We believe this model can be useful in a distributed product development context because the product is normally the main focus of work for the developers, and the shared product is something the developers can "talk about." Being kept updated about what is happening to the product will reduce the risk for double–work and integration problems. In addition, since the product is normally used throughout the project's life independent of the way the developers work and the way they are combined in groups, the product can be used as a permanent information basis for coordinating the work.

Our product awareness model will increase the quantity of product awareness, and will make it easy for developers to exchange this awareness information. Next section describes the details of the model. We emphasize that product awareness is only one of several forms of awareness in a product development environment [HG99]. Therefore the model introduced here should be used in combination with other models in order to support true simulation of physical proximity motivated in the introduction.

3.1 The Core Concepts of the Model

The product awareness model introduces a virtual space for exchanging awareness of activities involving a shared product. The model also supports *propagation* of awareness; awareness information can propagate from one part of the product to another, possibly through several intermediate parts. In this way, a group of developers working on one part of the product can get the proper awareness from any other part, in addition to the part they directly work with. This propagation property is important because of often large size and complex structure of the products being developed. The core concepts of the model are `product model`, `product object`, `awareness relation`, `direct awareness`, `mediated awareness`, `awareness producer`, and `awareness consumer`. These concepts and the associated relations are shown in the ER diagram in Fig. 3.

`Product model` is a representation of the shared product being developed by a group of developers. `Product model` constitutes the *shared space* where different parts of a product can register their *presence* in form of `product objects`. `Product objects` may represent documents, diagrams, source codes, etc. The product parts may already be stored in a CASE repository, on the Internet, or in a database elsewhere. For each such part, only one `product object` may exist in the `product model`, but the granularity of the parts is not limited by the awareness model. Each `product object` is capable of generating awareness information as a result of being manipulated.

The model supports a notion of *focus* for `product objects`. Each `product object`'s focus is defined by creating `awareness relations` from those `product objects` it focuses on, and to the `product object` itself. Each `awareness relation` has therefore a *source* and a *destination* `product object`, indicating the direction of the flow of awareness information. Criteria for deciding what each `product object` should focus on, i.e. how `awareness relations` should be created among `product objects`, is not defined by the model. `Awareness relations` may be defined based on product architecture, or other criteria such as the

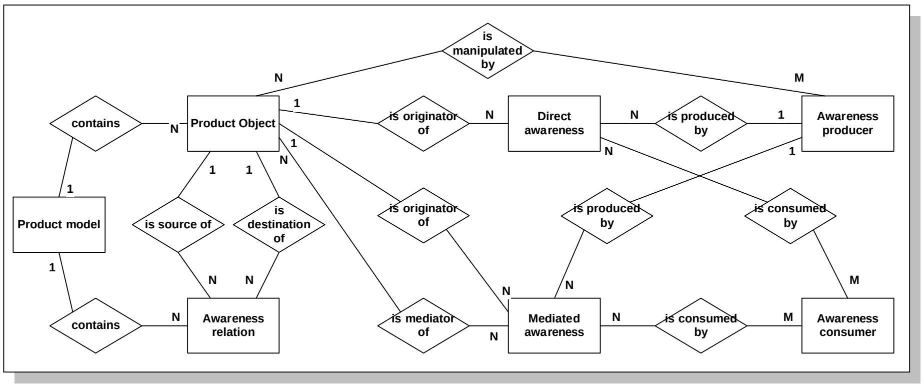

Fig. 3. The main concepts of the product–based awareness model and the relationships among them.

specific usage patterns of a project group. These criteria are mainly defined by the cluster and application layers in the framework shown in Fig. 1. The awareness model and its implementation only focus on providing easy mechanisms for creating `awareness relations` as needed.

A `product model` is in this way created in order to mirror the real product and the various relations among its parts. An example of a `product model` is shown in Fig. 4. It represents a product consisting of three main modules, *database*, *middleware* and *user interface*. In addition, each module consists of other sub–modules. In this example, `product objects` are created for each module or sub–module in the product. `Awareness relations` are created among the `product objects`, reflecting the paths through which one would like product awareness to be propagated. In this case an `awareness relation` exists from the database module to the middleware module, and one from the middleware module to the user interface module. For each of these modules, all the sub–modules are also connected to their parent module using awareness relations. This means, for instance, that if some object in the database module is modified, the middleware module will get awareness information about the modifications.

As the product development process proceeds, product parts may be manipulated in different ways. Manipulation activities include those that are normally supported by development tools, such as creating, accessing and deleting product parts. It is assumed that each manipulation of a product part is simulated in the `product model` by manipulating the part's corresponding `product object`. The various types of manipulation are not defined by the awareness model. Groups of developers can define and share their own manipulation types in the application layer of our framework (shown in Fig. 1). The awareness model represents developers and their tools as `awareness producers`. Each manipulation activity performed by an `awareness producer` will produce a unit of `direct awareness` in the `product model`. This information indicates a change in the

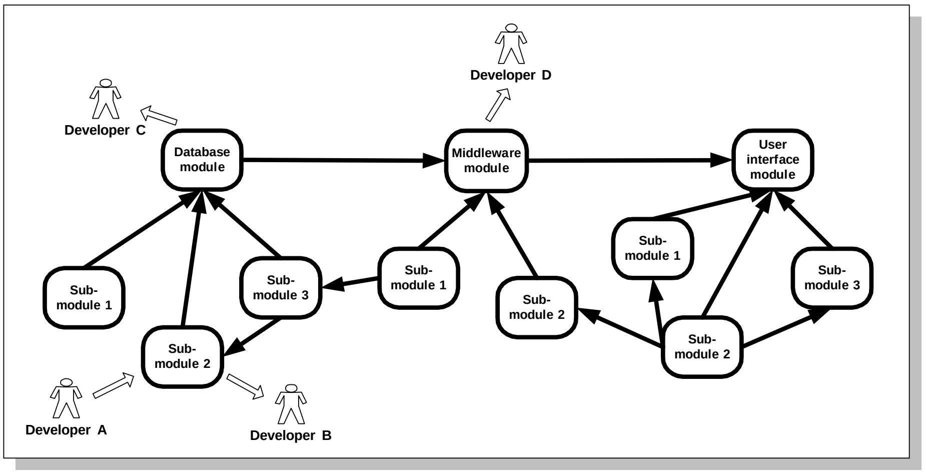

Fig. 4. An example Gossip product model. The rectangles are product objects and the arrows are awareness relations.

`product object`'s presence in the shared space. Each `direct awareness` contains information about the `awareness producer` who did the manipulation, the `product object` that was manipulated, an identifier for the manipulation type, and other user–defined values depending on the type of the manipulation.

Developers sharing the same `product object` can exchange `direct awareness` related to these objects. Moreover, `direct awareness` can be propagated to other `product objects`.Propagation of awareness inside the `product model` happens through the *mediation* mechanism. Each `product object` broadcasts all its awareness to all the `product objects` in the `product model`. Each `product object` that is focusing on the broadcasting `product object` will generate a new `mediated awareness` based on the broadcasted awareness. This `mediated awareness` contains the same information as the original awareness, except that its originator `product object` is changed to the receiving `product object`. However, the `mediated awareness` also contains a pointer to the originally manipulated `product object` (the "is mediator of" relation in Fig. 3). Mediation from a `product object` X to a `product object` Y is thus possible only if a path consisting of (possibly several) `awareness relations` exists from X to Y.

`Awareness consumers` represent developers or other applications that make use of the awareness generated by the `product model` and its `product objects`. In the framework of Fig. 1 `awareness consumers` are the clusters in the cluster layer, but awareness may also be consumed directly by any application. `Awareness consumers` have access to both `direct awareness` (produced by the `product objects` they work with directly) and `mediated awareness` (produced by other `product objects`, related to the `product objects` they work with directly). In Fig. 4 developer A is an `awareness producer` who is manipulating the product part presented by "Sub–module 2" in the `product model`. As a result of his manipulation, a `direct awareness` is generated. This `direct awareness` is used by developer B, who is an `awareness consumer` working with the same `product object`. In addition, the `direct awareness` is mediated by `product objects` "Database module" and "Middleware module." This causes `awareness consumers` C and D to receive a `mediated awareness` from their corresponding `product objects`. An actor (a developer or an application) can be both `awareness producer` (producing awareness related to own activities) and `awareness consumer` (consuming awareness of others' activities).

Using this product awareness model, each developer produces awareness information based on his own activities related to the shared product, for instance by manipulating a set of `product objects` in his workspace. Each developer can also have access to relevant product awareness by simply having the relevant `product objects` in his workspace. In addition, each unit of awareness information is directly connected to the developer who did the manipulation, in this way opening for social interaction among developers with related interests in the shared product. Another advantage of the model is that each developer only needs to focus on those `product objects` that are of direct importance to his own work, disregarding the rest. The `product objects` he is working with will inform him about peripheral changes in the product that might be of interest to him, at the same time hiding the irrelevant part of the awareness information. This can greatly reduce the effort needed for keeping an eye on everything that might be important to the developer's work.

4 Gossip: An Awareness Engine for Supporting Product Awareness

In the recent years the need for having generic awareness services has increased. A generic awareness service typically provides an interface for different kinds of clients to produce and consume awareness information in an easy way. Notification servers have been used widely as a technical solution for providing awareness services [RDR98]. Using a notification server, a client can generate events, which are captured by the server and redistributed to other clients. In addition, a notification server also has to decide, based on an awareness model, which clients need to receive which notifications. A notification server that implements an awareness model may be called an *awareness engine.* We have developed an awareness engine called Gossip for implementing the product awareness model described in the previous section. The functionality and the internal architecture of Gossip are described in the next sections.

4.1 The Functionality of Gossip

Gossip provides a set of uniform network–based services for creating and maintaining a product model in an evolutionary manner, and for delivering notifications based on the clients' activities related to this model. The product model

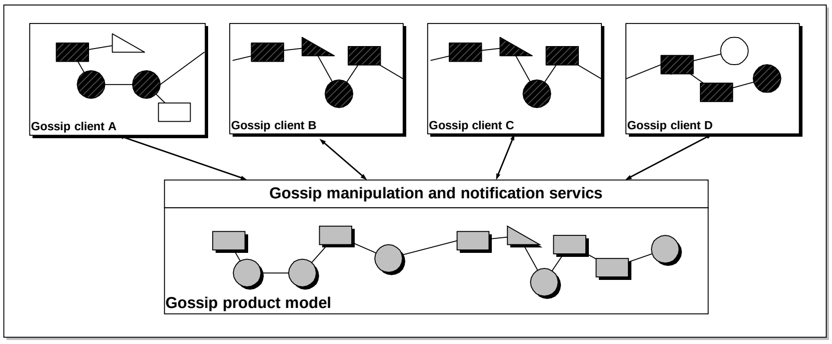

Fig. 5. Clients can use the services provided by Gossip to register their product objects and create awareness relations among them.

manipulation operations may be integrated into the various tools of the developers, making it possible to automatically maintain the product model while each developer is working with his local product part. In addition, a uniform notification delivery service makes it easy to recieve and consume notifications of other developers' manipulations.

Figure 5 shows a usage scenario for Gossip. Different clients communicate with Gossip in order to register a subset of their product parts (objects with black color) as product objects in Gossip (objects with gray color). In addition, awareness relations are created among the product objects registered in Gossip independently from which client owns which product objects. In this way, all the clients are integrated in a shared collaboration space based on a shared product model. Gossip enforces its own name space for the registered product objects, and implements mechanisms for keeping the product model consistent. The clients, on the other hand, are responsible for correct updates to the product model using Gossip's network protocol.

Manipulation mechanisms are accessible in form of *operations* on the shared product model. There are two groups of manipulation operations, *product object manipulation* and *awareness relation manipulation.* In addition, Gossip produces different *notification events* as result of product object manipulations. Query mechanisms are also provided to the clients in order to ask Gossip about various product model information. The following describes manipulation and notification functions provided by Gossip.

Product Object Manipulation. Clients can issue requests for product object manipulation. Besides product object registration and deletion, Gossip supports operations such as adding, changing, reading and deleting *attribute=value* pairs for each product object. The attribute sets for each product object type are defined by client groups, and will resemble the attributes of the real product parts that are to be shared within the client group. Note that product parts (e.g. the actual files, documents, diagrams, descriptive attributes, etc.) need not be stored in Gossip, but a pointer to each part can be registered. This policy requires that product manipulation operations precisely *simulate* the actual manipulations done on the product parts at each client site. This implies that the clients can still use their local repositories, and at the same time use Gossip to exchange product awareness. They will use product object manipulation operations to inform Gossip about how they manipulate their product parts. In this way Gossip will help its clients to keep updated about each other's activities related to a shared (distributed) product. Each client can choose to register in Gossip only a sub–set of its product parts (in Fig. 5, product parts with white color are not shared among clients).

Awareness Relation Manipulation. In addition to object manipulation, clients can also request awareness relation manipulation in order to create and manipulate awareness relations among existing product objects. Awareness relations are stored fully inside Gossip. Each awareness relation is represented in form of a set of *operation=strength* pairs, where *operation* decides what kind of awareness information the relation will mediate, and *strength* decides how many product objects each awareness information can be mediated through. For instance, if an awareness relation has an "updateAttribute" field with a strength of 2, the relation will mediate awareness related to all "updateAttribute" operations that have not already been mediated twice. Using operation=strength values for each awareness relation one can filter most of the information that is considered unnecessary. For instance, awareness of read operations on an attribute from a product object that is "three objects away" is normally considered not so important, while it would be interesting to get information about who is currently reading the objects in one's own workspace. Gossip supports operations for adding and deleting awareness relations, and adding, deleting, and updating operation=strength pairs on the existing awareness relations.

Notifications. Notifications are sent only for product object manipulations. There are two types of notifications, *direct* and *mediated.* Direct notifications have a pointer to the client who did the manipulation, and a pointer to the product object that was manipulated. Mediated notifications have in addition a list of pointers to all other product objects that the notification has passed through (recall that for mediated awareness, the manipulated product object is different from the product object that provides the awareness, as shown in Fig. 4). Additional information can be contained in each notification event for allowing the receiving client synchronize its own state. For instance, a "createNewObject" notification will contain the identifier of the new product object, and an "updateObject" notification will have the name and the new value of the updated attribute.

The set of available operations is extendable. In particular, there is no limit on which attributes can be manipulated for each product object, and the manipulation types that can be simulated are not predefined. All this can be decided among the clients sharing the product model. Gossip will only propagate the awareness and produce the proper notification events.

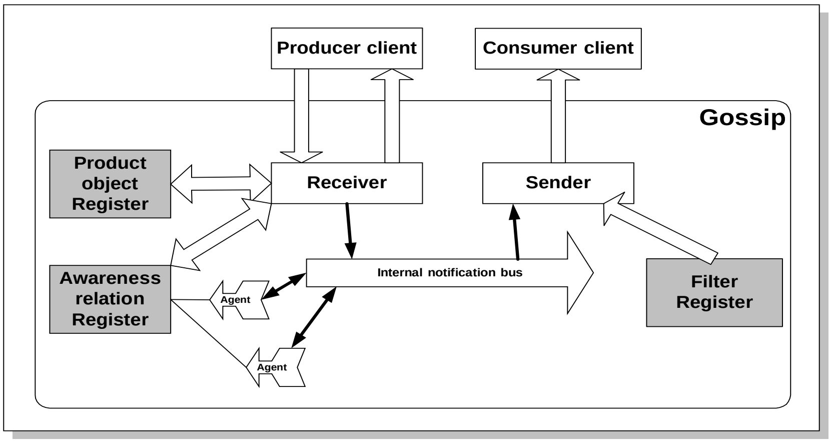

Fig. 6. The internal architecture of Gossip.

4.2 The Architecture of Gossip

Figure 6 shows the internal architecture of Gossip. There are two types of Gossip clients. *Producer clients* can send requests to Gossip for performing an operation on the product model (product object or awareness relation manipulation). The product model is stored in two registers, *product object register* and *awareness relation register*. Each product object manipulation operation may change the product object register, and will eventually generate one or more notification events (awareness relation manipulations do not produce notifications). *Consumer clients* can receive notification events from the server. A client can choose to be a producer, a consumer, or both. When a request is sent to Gossip by a producer client, *Receiver* receives the request and is responsible for performing the requested operation on the product model. After an operation is performed, the producer client will get an acknowledgement event, and a new direct notification event is created. This direct notification is then sent to an internal *notification bus*.

Each awareness relation is implemented in Gossip in form of an *awareness agent* with a source and a destination product object. Awareness agents are responsible for generating mediated notification events on behalf of their destination product objects, and therefore listen to the internal notification bus to monitor product object manipulations. Each awareness agent will check the notification bus for events that have originated from the agent's source product object. For each such event, the agent will generate a new mediated notification event on behalf of its destination product object. These mediated notification events are again sent to the notification bus, and other awareness agents may in turn generate new events based on them.

Sender receives all the notification events from the notification bus. It then checks each event against a *filter register* before sending them out. Each consumer client has the possibility for registering a filter that will prevent certain events from being sent to it. The filter uses a set of criteria for filtering out events. One important such criteria is to check for the owner of the event. Different clients, or groups of clients, may create their own awareness relations in the shared product model, in this way specifying a particular configuration of awareness information that is needed by that client or group. Each mediated notification event therefore contains a field indicating which such configuration the event belongs to.

4.3 The Implementation of Gossip

Gossip is implemented in the Java programming language. The server provides a small set of public interface classes. These classes hide most of the network–related details from clients. Making the interface as simple as possible has been one of our main goals in order to motivate developers to develop Gossip–enabled clients, or extend existing clients with Gossip–related functionality. Using the interface classes the clients can easily send requests to, and receive notifications from Gossip.

Network communication in Gossip, both towards clients and internally in the notification bus, is based on JSDT[1] (Java Shared Data Toolkit). JSDT is a flexible toolkit provided by Sun as an extension to the Java Development Toolkit. This toolkit implements useful groupware abstractions such as sessions and channels. The connection between the clients and Gossip is through a JSDT session. Inside this session, two channels are used for communication between Gossip and the producer and consumer clients.

The internal notification bus is implemented in form of a network channel that is accessible only by other Gossip servers. Several Gossip servers can share the same notification bus, in this way creating a Gossip network. This is useful for scalability and performance reasons. Our experience with using Gossip shows that inside a high–speed local network notifications are distributed in real time. We have in fact used Gossip in a synchronous graphical group editor where the notifications are used to synchronize the screens of the clients. The notifications are distributed in a much slower speed on the Internet. Having one local server for each local group can help to build a more optimized Gossip infrastructure.

5 Conclusions

In this paper we have discussed the importance of awareness in product development projects where the developers cooperate across geographical distances for long periods of time. We have discussed product awareness, and have introduced an awareness model that puts focus on the shared product and the propagation of awareness in large composite products. This awareness model is implemented

[1] http://java.sun.com/products/java-media/jsdt/

in form of an awareness engine called Gossip, which is used as a part of a framework for collaboration support in distributed product development. The design and implementation of Gossip are described.

In addition to Gossip, and as a part of the discussed framework, we have implemented two test applications. One is a graphical editor for creating simple Entity–Relation models. This editor is used to demonstrate the capability of Gossip to support synchronization of screens in real time. The editor also demonstrates how the various operations in a tool can be configured to automatically trigger operations in Gossip. The other application is a shared workspace application that demonstrates how different awareness models can be used in combination.

There are several issues that constitute our future research and development agenda. One important issue is privacy. The high specialization of labor among developers makes it unrealistic to expect that they will expose information about their activities without a second thought. The name "Gossip" is chosen deliberately to emphasize this. Gossip provides easy mechanisms for inserting and removing product parts from a shared space, but more advanced mechanisms, such as access control, are needed.

The current version of Gossip does not fully implement the filtering mechanisms discussed in the previous section. We need more usage data in order to find out what filter mechanisms and criteria are needed. With large scale deployment of Gossip, filter mechanisms will become a necessity not only for having higher degree of individual tailoring, but also because of technical issues involved regarding network bandwidth usage.

Acknowledgements

I thank Arne Sølvberg, Monica Divitini, and the anonymous reviewers for their comments on this paper. Part of this research is supported by a grant from Andersen Consulting Forskningsfond, Norway.

References

BF93. Steve Benford and Lennart Fahlèn. A Spatial Model of Interaction in Large Virtual Environments. In *Proceedings of ECSCW, Milano, Italy*, pages 109–124, September 1993. Kluwer Academic Publishers.

BHI93. Sara Bly, Steve R. Harrison, and Susan Irwin. Media Spaces: Video, Audio, and Computing. *Communications of the ACM*, 36(1):28–47, January 1993.

BJ75. Frederick P. Brooks Jr. *The Mythical Man–Month – Essays on Software Engineering*. Addison–Wesley, Reading, MA, 1975.

CGN95. Erran Carmel, Loey F. George, and Jay F. Nunamaker, Jr. Examining the Process of Electronic JAD. *End User Computing*, 7(1):13–22, 1995.

Con68. Melvin E. Conway. How do committees invent? *Datamation*, 14(4):28–31, April 1968.

CSC98. CSCW'98, editor. *Proceedings of the Conference on CSCW, Seattle, Washington, USA*, November 1998. ACM Press.

DB92. Paul Dourish and Victoria Bellotti. Awareness and Coordination in Shared Workspaces. In *Proceedings of the Conference on CSCW, Toronto, Canada*, pages 107–114, October 1992. ACM Press.

DLN97. Douglas L. Dean, James D. Lee, and Jay F. Nunamaker, Jr. Group Tools and Methods to Support Data Model Development, Standardization, and Review. In *Proceedings of the 30th Hawaii Int'l Conf. on System Sciences*. IEEE Computer Society Press, 1997.

FPBP95. Ludwin Fuchs, Uta Pankoke-Babatz, and Wolfgang Prinz. Supporting Cooperative Awareness with Local Event Mechanisms. In *Proceedings of ECSCW, Stockholm, Sweden*, pages 247–262. Kluwer Academic Publisher, September 1995.

GG98. Carl Gutwin and Saul Greenberg. Effects of Awareness Support on Groupware Usability. In *Proceedings of CHI, Los Angeles, CA, USA*, pages 511–518, April 1998. ACM Press.

Gri98. Rebecca E. Grinter. Recomposition: Putting It All Back Together Again. In CSCW'98 [CSC98], pages 393–402.

HG99. James D. Herbsleb and Rebecca E. Grinter. Splitting the Organization and Integrating the Code: Conway's Law Revisited. In *Proceedings of ICSE'99, Los Angeles, California, USA*, May 1999. ACM Press.

HW98. Chung-Hua Hu and Feng-Jian Wang. A Multi–User Visual Object–Oriented Programming Environment. In *Proceedings of COMPSAC'98, Vienna, Austria*, pages 262–268, 1998. IEEE Computer Society Press.

JMR92. Matthias Jarke, Carlos Maltzahn, and Thomas Rose. Sharing Processes: Team Coordination in Design Repositories. *Intelligent and Cooperative Information Systems*, 1(1):145–167, March 1992.

KCI87. Herb Krasner, Bill Curtis, and Neil Iscoe. Communication Breakdowns and Boundary Spanning Activities on Large Programming Projects. In *Proceedings of Empirical Studies of Programmers, Washington D.C., USA*, pages 47–64. Ablex Publishing Corporation, December 1987.

KE88. Robert Kraut and Carmen Egido. Patterns of Contact and Communication in Scientific Research Collaboration. In *Proceedings of the Conference on CSCW, Portland, OR, USA*, pages 1–12. ACM, September 1988.

KS95. Robert E. Kraut and Lynn Streeter. Coordination in Software Development. *Communications of the ACM*, 38(3):69–81, March 1995.

MKF$^+$97. Tim Mansfield, Simon Kaplan, Geraldine Fitzpatrick, Ted Phelps, Mark Fitzpatrick, and Richard Taylor. Evolving Orbit: a progress report on building locales. In *Proceedings of Group'97, Pheonix, USA*, pages 241–250, November 1997. ACM Press.

Par72. D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

RDR98. Devina Ramduny, Alan Dix, and Tom Rodden. Exploring the design space for notification servers. In CSCW'98 [CSC98], pages 227–235.

RG96. Mark Roseman and Saul Greenberg. TeamRooms: Network Places for Collaboration. In Proceedings of the Conference on CSCW, Cambridge, Mass., USA, pages 325–333, November 1996. ACM Press.

TW97. Hilda Tellioğlu and Ina Wagner. Negotiating Boundaries – Configuration Management in Software Development Teams. *Computer Supported Cooperative Work*, 6(4):251–274, 1997.

VS95. Iris Vessey and Ajay Paul Sravanapudi. CASE Tools as Collaborative Support Technologies. *Communications of the ACM*, 38(1):83–95, 1995.