

Distributing Knight

Using Type-Based Publish/Subscribe for Building Distributed Collaboration Tools

Christian Heide Damm and Klaus Marius Hansen

Department of Computer Science,
University of Aarhus,
Åbogade 34, 8200 Aarhus N, Denmark
{damm,marius}@daimi.au.dk

Abstract. Distributed applications are hard to understand, build, and evolve. The need for decoupling, flexibility, and heterogeneity in *distributed collaboration tools* present particular problems; for such applications, having the right abstractions and primitives for distributed communication becomes even more important. We present *Distributed Knight*, an extension to the *Knight* tool, for distributed, collaborative, and gesture-based object-oriented modelling. Distributed Knight was built using the *type-based publish/subscribe* paradigm. Based on this case, we argue that type-based publish/subscribe provides a natural and effective abstraction for developing distributed collaboration tools.

1 Introduction

Distributed collaboration tools support multiple users in cooperating, coordinating, and communicating distributed in space and possibly also in time. Building such distributed collaboration tools is, however, hard: partial failures have to be taken into account, the tool must be operatable at interactive speeds, and replicated data must remain consistent (Pra99).

What kind of distributed communication abstractions are appropriate then? It is commonly recognized that event-based communication, providing loose coupling, is appropriate for open-ended, distributed, and co-located integration of applications (BCTW96). Event-based communication is centered around the events that are passed around between clients using implicit addressing. This means that clients do not send messages directly to the other clients, and they are thus not dependent on the other clients' interfaces. The *Field Environment* (Rei90) is an example of using event-based communication to integrate tools. The *Design Environment* (Wik98) uses a mixture of central and replicated data, where the replicated data is kept consistent during synchronous collaborative work using event-based communication.

This paper examines and discusses the *type-based publish/subscribe (TPS)* event-based distributed communication abstraction (Eug01) in the context of dis-

tributed collaboration tools and argues that this abstraction may provide a natural approach to tool integration, particularly in the context of distributed and heterogeneous clients.

The domain of software development environments is an example of an application domain where distributed collaboration capabilities are becoming more important, due to the increasing globalisation also of software development. It is thus appropriate to examine the *Knight* tool, which is object-oriented modelling tool, and specifically how we implemented a distributed version of the Knight tool, *Distributed Knight*, based on type-based publish/subscribe. This case study is used to discuss many of the common issues of building distributed collaboration tools, such as handling data replication, evolution, and scalability, from the perspective of type-based publish/subscribe.

1.1 Structure of This Paper

The rest of this paper is structured as follows: Section 2 introduces the publish/subscribe abstraction for distributed communication with emphasis on type-based publish/subscribe. Based on this introduction, we hypothesise that type-based publish/subscribe is effective for building distributed collaboration tools and devote the rest of the paper to the discussion of that hypothesis. Section 3 presents the case study of turning the Knight tool into Distributed Knight. Following that, Section 4 discusses the hypothesis in connection to our case study. Finally, we discuss related work, future work, and conclude.

2 Type-Based Publish/Subscribe

Publish/subscribe is an event-based communication style in which *publishers* publish events, and *subscribers* subscribe to and receive the events they are interested in.

The traditional publish/subscribe systems are based on the *subject-based* (or *topic-based*) variant of publish/subscribe (TIB99; Tal99; AEM99). In subject-based publish/subscribe, events are published to particular named subjects, and subscribers can then subscribe to subjects and will receive all events that are published to the given subjects.

The *content-based* publish/subscribe variant of publish/subscribe is a more dynamic variant, in which a subscriber can specify the runtime properties that events should have (ASS⁺99; CRW00; CNF98). Only the events that satisfy these properties will be delivered to the subscriber.

Type-based publish/subscribe (TPS) (Eug01) is a recent *object-oriented* variant of the publish/subscribe interaction style. In TPS, events are *objects*, i.e., instances of native types. A subscriber of a particular type of objects will only receive instances of that type and its subtypes. Subscriber-specified *content filters* further limit the events that will be delivered to the subscriber. Content

filters are specified in the native language based on the events' public attributes and methods, and they may be processed remotely to reduce network load.

The original motivation for introducing *type-based* publish/subscribe was that it (potentially) supports type safety and encapsulation. Both of these are considered important factors in producing robust, error-free stand-alone applications. It seems obvious that it is even more desirable to ensure these properties also in distributed applications, where the complexity is even higher than in stand-alone applications.

2.1 Implementation and Example

The type-based publish/subscribe variant has been implemented for Java (EG01; EGS01), and it was recently proposed to extend Java to directly support TPS (Java_{PS}, (EGD01)). It has been shown that Java_{PS} enables better support for TPS than Java does (DEG02).

We have chosen to implement TPS servers and clients in the Itcl language (McL93). However, due to the interpreted nature of Itcl and its powerful reflection mechanisms, it is not necessary to actually extend Itcl in order to obtain the advantages that Java_{PS} offers over Java. We will not go any further into the implementation of TPS, since the focus of this paper is the usefulness of TPS in implementing distributed collaboration tools.

Figure 1 shows an example of an event type from Distributed Knight (using Java_{PS} syntax because it is easier to read than Itcl for most people). The *MouseEvent* is an awareness event, and it contains the information that a given user has moved/pressed/... the mouse in a given place. Note that event types can contain private fields, and subscribers must express content filters based on the public fields. Also shown in Figure 1 are examples of how clients publish and subscribe to MouseActionEvents.

2.2 Hypothesis

Based on using type-based publish/subscribe to build distributed collaboration tools, the rest of this paper is concerned with investigating the hypothesis that type-based publish/subscribe makes it easier to

- understand problems and potential solutions,
- create solutions, and
- evolve existing and new systems

for distributed collaboration tools compared to other object-based communication abstractions such as RMI as well as to other publish/subscribe abstractions.

Understand Problems and Potential Solutions. One of the main benefits of the object-oriented paradigm is that it provides a conceptual framework for

```
// Defining MouseActionEvents
public class MouseActionEvent extends SessionAwarenessEvent {
    private int actionType;
    private int diagramID;
    private float x;
    private float y;
    public int getActionType() {return actionType;}
    ...
    public MouseActionEvent(int userID, ..., int actionType, ...) {
        SessionAwarenessEvent(userID,sessionID);
        this.actionType = actionType;
        ...
    }
}

// Publishing MouseActionEvents
MouseActionEvent event = new MouseActionEvent(userID,
    sessionID, ButtonPress, diagram.ID, mousePos.x, mousePos.y);
publish event;

// Subscribing to MouseActionEvents
Subscription s = subscribe (MouseActionEvent event) {
    // content filter
    if(event.getSenderID() != PublishSubscribe.clientID) {
        // only accept mouse actions in our session
        return (event.getSessionID() == sessionID);
    } else {
        // ignore our own mouse actions
        return false;
    }
} {
    // event handler
    ... indicate the mouse action in the user interface ...
}
s.activate();
```

Fig. 1. Publishing and Subscribing to Events

all of system development, i.e., it allows software developers to create implementations based on an understanding of the problem and solution domain. Object-oriented programming languages minimise the translation of this understanding to the actual implementation and back, because they provide abstractions that fit well with how developers may perceive and understand the world.

Traditional publish/subscribe systems that implement subject-based or content-based publish/subscribe force the developer to represent his understanding in a primitive way. The same problem is evident when using relational databases to store object structures. For both these cases, it would, on a low level, be much

easier to read and understand code that is written *without* a translation from one paradigm to another.

Create Solutions. Since TPS does not force a translation away from the object-oriented paradigm, it becomes easier to express an “abstract” solution in the programming language. Concrete examples of this include that the developer publishes *objects* without worrying about the serialization, and that the developer can express content filters in the native object-oriented language instead of in a declarative special-purpose language.

Evolve Existing and New Systems. Due to the inherent decoupling in publish/subscribe systems, it is easy to introduce new applications to a system: event-based communication is inherently many-to-many, and by focussing on sending events, the strong interface coupling of RMI-like systems is avoided. TPS may furthermore support reusable libraries or frameworks of, e.g., event types and subscriptions (SA97).

3 The Case of Distributed Knight

This section presents the Knight tool briefly and discusses how it was extended with distributed collaboration capabilities using type-based publish/subscribe.

3.1 The Knight Tool

The Knight tool (DHTT00a) was originally conceived of as a tool to support *co-located* collaborative modelling using the Unified Modeling Language (UML, (OMG01)). Based on observations of modelling practice, we designed and implemented a tool that supports the kind of modelling work that is usually performed on traditional whiteboards. Figure 2, left, shows the Knight tool in use on an *electronic whiteboard*. An electronic whiteboard has a large, pressure-sensitive surface that displays a computer screen and on which users may draw using pens. Alternatively, Knight may be used with more traditional input devices such as mice or track balls. To support an interaction much like that on an ordinary whiteboard, Knight uses *gestures* to create, delete, and modify most elements. Figure 2, right, shows an example of creating a class using a gesture. The tool has been implemented in a mixture of Tcl (McL93), C++, and C.

The Knight tool has subsequently been commercialized by Ideogramic ApS as *Ideogramic UML* (<http://www.ideogramic.com/products/uml/>).

3.2 Distributed Knight

Distributed Knight extends Knight by allowing people to collaborate on UML models even though they are geographically apart. The current status of Distributed Knight is that any number of clients are able to work collaboratively

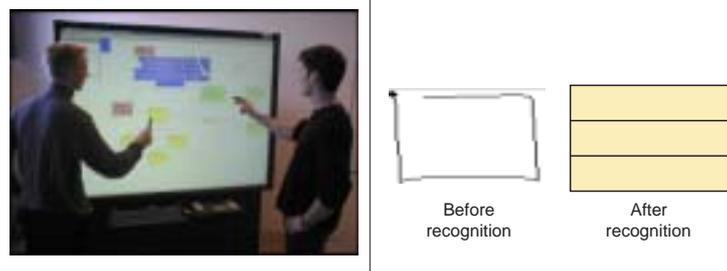


Fig. 2. Left: Use of Knight on an Electronic Whiteboard. Right: Gesture Recognition in Knight

on UML models. We have also implemented awareness of, e.g., who is joining and leaving sessions and indications of where the other users are working etc. All this has been implemented using type-based publish/subscribe as detailed below.

3.3 Implementing Distributed Collaboration

A *Distribution* component in Knight contains most of the distribution functionality. The Distribution component is surprisingly isolated from the rest of Knight and has required only minor changes to the existing components.

Propagating Changes in the Repository. Each stand-alone Knight instance contains a *Repository* component that contains model and diagram data for that instance. In Distributed Knight, the Repository objects are replicated for each client of a distributed session, and changes to the Repository must therefore always be propagated to all the other clients in order to maintain consistency. The replication is necessary to achieve the responsiveness expected for an interactive application (Pra99).

The Observer (GHJV95) structure that is used internally in Knight is ideal for discovering changes to the model elements in the Repository. The Distribution component registers itself as an Observer on the Repository and will be notified whenever a model element has been created, changed, or deleted.

The Distribution component also observes the command history of Knight, so that it knows when a Command (GHJV95) has been executed or unexecuted. This is important because Distributed Knight publishes Repository changes according to Commands. A Command may create/change/delete any number of model elements, and these changes should be published to the other clients as one unit. For example, when the user creates an Association, two AssociationEnds are automatically created but not shown, and it does not make sense to publish the Association without the AssociationEnds.

In summary, the Observer on the Repository tells us which elements have changed, and the Observer on the command history tells us how to group the changes together and publish them to the other clients. In this way, the same kind of natural, event-based communication *within* Distributed Knight is used for communication *between* Distributed Knight clients. This mainly accounts for the fact that building Distributed Knight required little changes to Knight itself.

The actual propagation of changes to other clients is done by publishing *SessionDataEvents* (Figure 3). A *SessionDataEvent* contains a list of data changes, each representing either a created, changed, or deleted element. In order to create an element, only the type and unique ID of the element are needed, where the type is the qualified class name, such as `Foundation.Core.Association`. Once an element has been created, it can be changed any number of times. A change involves all the information described above, i.e., the internal data of the element and the context of the element. Finally, an element can be deleted.

The next section will explain the event hierarchy in more detail.

The Event Hierarchy. All events are automatically equipped with the unique ID of the publisher (accessible through the `getSenderID` method, cf. Figure 1). This is useful when a publisher is also a subscriber on the same event types, since in many situations, the subscriber should ignore the events it has published itself.

The *UserEvent* basically keeps track of which user creates an event (Figure 3). As shown, events are responsible for implementing *getData* and *setData* methods for serializing and deserializing. These methods are not necessary in a language such as Java that provides default serialization and deserialization.

Events for actual collaboration in Distributed Knight are all connected to sessions in which users participate (*SessionEvent*). An example of non-session events are *InstantMessagingEvents* used by our instant messenger client for, e.g., sending chat messages between clients (see Section 4.2).

SessionManagementEvents handle invitations to sessions, requesting lists of active sessions, joining and leaving sessions, etc.

The *MouseEvent* presented above (Figure 1) is an example of a *SessionAwarenessEvent* that is used to give awareness of other users' actions.

4 Discussion

Generally, type-based publish/subscribe has worked well for implementing Distributed Knight, especially in terms of decoupling and ease of development. Below, we discuss advantages and disadvantages on aspects of using TPS.

4.1 Integration into the Object-Oriented Paradigm

Type-based publish/subscribe allows developers to work with normal types and objects from the native object-oriented language. This is in contrast to the

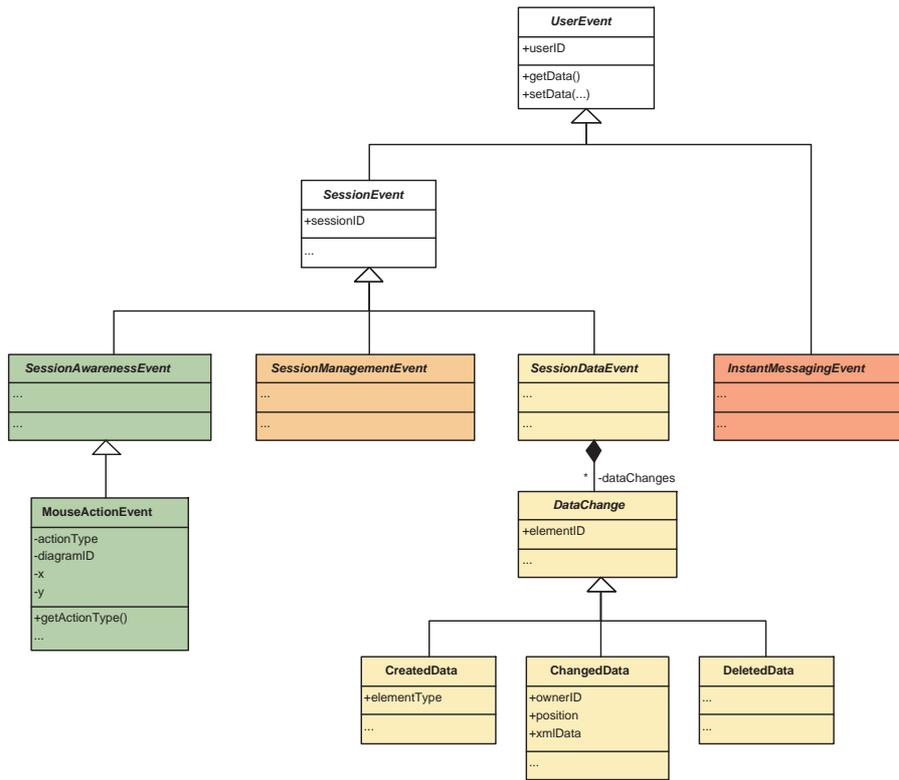


Fig. 3. Part of the Distributed Knight Event Hierarchy

traditional subject-based and content-based variants of publish/subscribe that force the developer to shift to a more primitive communication level based on name/value pairs. By avoiding this shift, TPS makes it faster to develop distributed applications, and the result is easier to understand and maintain. The shift is also avoided with respect to content filters. With TPS, these can be expressed in the native language, but with the traditional publish/subscribe variants, they are typically expressed in a string format in a way that is similar to how SQL queries are expressed.

The claim that using native language constructs is beneficial is supported by considering how remote procedure calls (RPC) and more lately remote method invocations (RMI) made it much easier to develop distributed applications. Like TPS, RPC and RMI also avoid forcing the developer to worry about the details of serialising/deserialising objects, transmitting the data, error handling, etc.

The notion of *event hierarchy* is a natural successor to the nested subjects in subject-based publish/subscribe, even though the event hierarchy does not have to be strictly hierarchical. It is our experience that the event hierarchy provides a natural coarse-grained filtering. A client that is only concerned with session management and awareness (such as the `AwareMessenger`, see Section 4.2) should ideally be able to ignore the part of the event hierarchy that is related to session data, and this is accomplished simply by only subscribing to the relevant parts of the event hierarchy.

TPS allows the developer to *model* the “communication mechanisms” of an application, just like one can model a problem domain or an application in general. This model is reflected in the event hierarchy, which may then, e.g., be reused across applications.

Our event hierarchy has so far only been used to implement one distributed application, namely Distributed Knight, but large parts of the event hierarchy are actually application independent. For example, the session data events are not specific to Distributed Knight, because they can be used for any application that implements an object-based model. Similarly, the session management events are also not specific to Distributed Knight. In this respect, the event hierarchy may actually evolve into a general distribution library – something that is hard to imagine for the traditional publish/subscribe variants. This in fact further emphasizes the strong relation of TPS to the object-oriented paradigm, through a principle of code reuse.

As will be described in Section 4.2 below, it is awkward to use publish/subscribe in some situations. Sometimes, a stronger coupling between two clients makes RMI a more natural communication abstraction. It is our experience that TPS and RMI complement each other nicely: TPS can be used to publish information to a wider audience, and events may optionally include an RMI reference as an attribute of the event. If one or more of the subscribers wish to go into a closer, more coupled collaboration with the publisher, this can be done through the RMI reference.

Another problem with publish/subscribe in general is the lack of high-level structuring mechanisms for event-based systems. In our case, we might like our instant messengers (see Section 4.2) not to receive events only relevant to Distributed Knights and vice versa. This can be remedied, however, using *scopes* (FMMB02), a module mechanism developed for event-based systems. We plan to investigate and possibly integrate such mechanisms in our type-based publish/subscribe architecture in the future.

4.2 Decoupling

We previously integrated the Knight tool with the Rational Rose tool (<http://www.rational.com/rose/>) based on Microsoft COM (Rog97). This integration provided a general architecture for integrating tools with compatible meta-models based on a set of requirements for the COM interface of such tools (DHTT00b). It is indeed possible to integrate tools in that way, but the result seems to be that the tools become strongly coupled to each other. It, e.g., requires a big effort to add a new client type due to the very different interfaces of existing tools.

Using publish/subscribe, on the other hand, results in *space decoupling*, since publishers do not know the location of subscribers and vice versa. This has been useful for creating new client types without modifying existing client types: take as an example our implementation of *AwareMessenger*, a context-aware instant messaging client (Figure 4). Apart from being an ordinary instant messenger

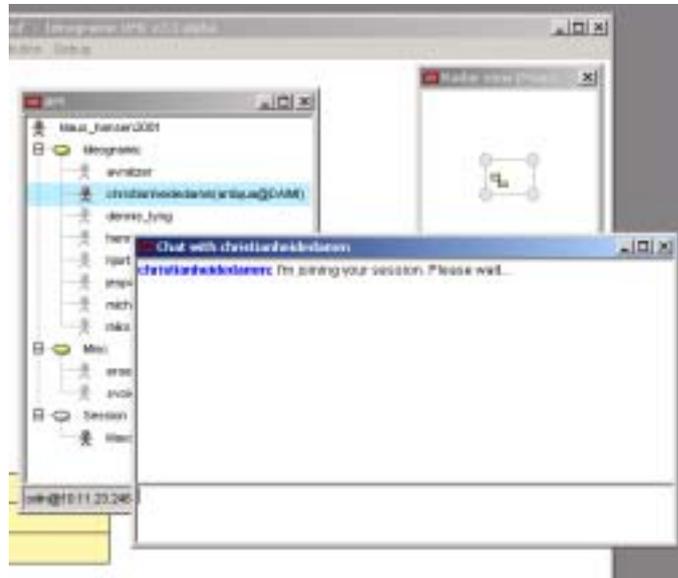


Fig. 4. AwareMessenger

(implemented using TPS), AwareMessenger also subscribes to join session and leave session events. In this way, potential collaborators get awareness of which sessions are ongoing, and they might want to join a relevant session. The figure shows a concrete example of this: since *klaus_hansen2001* has started a session (and invited *christianheidedamm*), a transient *session group* is created in AwareMessenger. AwareMessenger also provides a user interface for creating, joining, and inviting to sessions.

Other types of clients that we are considering implementing and that would be both useful and require only moderate effort include a code generation client and a “ticker tape” application showing digests of the actions made in sessions.

The *flow decoupling* of publishers and subscribers helps in achieving interactive performance in a user interface-oriented environment such as Distributed Knight. Publisher publish events without waiting for results and subscribers receive data in the form of events without explicitly waiting for data. This puts extra constraints, however, on either a server or individual clients to handle ordering and dependencies between distributed commands.

Decoupling can also be problematic in some respects: for the initial synchronization of a client joining a session, exactly one client already in the session should send its data to the joining client. This is awkward using publish/subscribe alone, and we solve it by combining publish/subscribe with RMI as described in Section 4.1: publish/subscribe is used to discover the location of one of the session clients, and RMI is used for transferring data from that location.

4.3 Performance and Scalability

Many events need to be *pruned* for performance reasons. One example is the `MouseEvent`s: clients may produce a very large number of such events as the result of user actions, and it is inefficient to publish all of them as events, since the handling of the events involves updating a display.

Another example, this time taken from the `SessionDataEvents`, is when the user moves an element in the user interface. During the move, the element will be changed many times, but it is inefficient to publish all the intermediate states. Instead, we prune all but the last `ChangedData` object.

It is important to note that pruneable events must be idempotent; in the case of the `MouseEvent`, this means that absolute mouse positions are transmitted instead of deltas.

A promise of type-based publish/subscribe is scalability, based among other on the use of remote content filtering. Our current implementation of publish/subscribe evaluates content filters locally at the subscriber, mainly for simplicity reasons. It can be argued that this is not a problem for an interactive application such as Distributed Knight: sessions and servers will serve only a small number of users in our settings, and except for initial synchronization only small amounts of data are transferred.

If we were to implement, e.g., streaming audio and video using publish/subscribe, then this should *not* be filtered locally at the subscriber due to the large data volume.

The current implementation of TPS is based on a central server, again mainly for simplicity reasons. The Java implementation of TPS described in (Sar01) is implemented using IP multicast, with one multicast group per event type, and this is presumably much more efficient than our server-based implementation.

5 Related Work

A major part of the research presented in this paper has been to provide the first, large-scale use of TPS. Previous examples of use, as in (Eug01), have been small examples, even though other variants of publish/subscribe have been used for large systems. We have obviously built upon the research of Eugster et al., but our major contribution with respect to TPS is to discuss *how* and *why* TPS may be used for distributed collaboration in interactive systems.

The *Field Environment* (Rei90) provides an early example of a tool integration effort. Field is a software development environment with code editor, compiler, debugger, etc., and it is based on a so-called *selective broadcast* mechanism – basically a simple content-based publish/subscribe system.

Even though selective broadcast is the only distributed communication abstraction, Field makes heavy use of RPC-like interaction: The broadcast implementation also supports synchronous command invocation, where the “client” is blocked until all “subscribers” have processed the command and returned a result. This kind of interaction resembles *group communication* (Par92). TPS does not have the division between asynchronous and synchronous publishing, but instead integrates seamlessly with RMI to support one-to-one, synchronous communication.

It is one of the main criteria in the design of the Field Environment that “tools must be able to interact with each other directly” (Rei90), e.g., when the editor needs the debugger to set a break point. It is clearly the case that the components in an integrated environment need to be aware of each other and to a certain extent collaborate directly with each other. On the other hand, it is our belief that the individual components should be designed in a way that makes them as independent of the other components as possible. This gives a more robust and flexible system. It may be the case, then, that it is better to expose the necessary dependencies by using explicit, synchronous communication as with RMI, than it is to merge it with the normal publish/subscribe communication.

Groove is an example of a commercial peer computing platform “designed to host peer-to-peer applications and business solutions” (<http://www.groove.net/>). From the point of view of use scenarios, it shares many of the same goals as Distributed Knight. Technically, however, it appears to be based on message

queues for the distribution of data. Even though Groove appears to be working on less structured data than Distributed Knight, its apparently open architecture may provide an interesting way of experimenting with multimedia capabilities in a Distributed Knight.

6 Status and Future Work

Distributed Knight has been implemented in a prototype version as presented in this paper and is actively being used by members of our group for distributed object-oriented modelling. Currently only class diagrams are supported, but the largest current problem is probably our (very) optimistic concurrency control. This will be one of the first areas of further research: many solutions to conflict resolution and ensuring consistency have been proposed for group editors and we will investigate and implement some of these. However, we expect the actual implementation, and suitable solutions, to be dependent on the hierarchical nature of UML models. It would be possible to take advantage of this, e.g., to provide a relatively fine-grained determination of which actions are in conflict with each other based on the hierarchical structure of UML.

Studies of use of Distributed Knight are ongoing. We have conducted interviews with developers on their distributed collaboration practices and in particular their use of instant messaging technologies for this. Based on personal experience and this insight, the AwareMessenger has been designed. The setup with Distributed Knight and AwareMessenger has currently been rolled out in prototype versions to allow distributed collaboration between Mejlgade, Aarhus (where Ideogramic ApS is located) and Aabogade, Aarhus (where the authors of this paper are located).

Even though use studies are just starting, it is evident that there is a need for increased awareness and session management support: audio and video links are something that will be essential to provide context awareness between participants. The current plan is to build this into AwareMessenger so that it will support *levels of coupling* between participants ranging from chatting without having a common session to having a common session with full audio and video links between participants.

Part of the promise of using standards such as UML is that of interoperability. We plan to extend an existing CASE tool, Rational Rose, with a component that is able to subscribe to TPS events and make it handle, in particular, data events. This is quite feasible since Rational Rose implements the UML standard and has an extensive extensibility interface. Extending Rational Rose is an example of using type-based publish/subscribe to build collaboration facilities involving different tools with the same application domain and compatible metamodel. This will, potentially, lead to a framework based on our type-based model of communication for integration of tools in other application domains. The Ideogramic Ideas (<http://www.ideogramic.com/products/ideas/>) tool for general collaborative diagramming will provide an obvious test for such as framework that will also include more “general” clients as the messenger described above.

Lastly, and technically, we plan to look further into issues of scalability, conflict resolution, and security in order to ultimately create a commercial Distributed Knight.

7 Summary

This paper has discussed using the type-based publish/subscribe paradigm for distributed communication to implement distributed collaboration tools.

Based on the case of the Distributed Knight extension of the Knight tool for co-located, collaborative modelling, we have argued that TPS is well-suited for implementing such tools.

In particular, we have provided initial evidence of our hypothesis that using type-based publish/subscribe, makes it easier to

- understand problems and potential solutions,
- create solutions, and
- evolve existing and new systems

in the context of distributed collaboration tools.

8 Acknowledgements

The work presented in this paper has been partly supported by the Centre for Pervasive Computing (<http://www.cfpc.dk/>). Thanks also to the users and co-developers of Knight and Ideogramic UML.

Bibliography

- [AEM99]M. Altherr, M. Erzberger, and S. Maffeis. iBus - a software bus middleware for the Java platform. In *Proceedings of the Workshop on Reliable Middleware Systems of IEEE SRDS'99*, pages 43–53, 1999.
- [ASS⁺99]M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *Proceedings of ACM PODC'99*, pages 53–61, 1999.
- [BCTW96]D.J. Barrett, L.A. Clarke, P.L. Tarr, and A.E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, 1996.
- [CNF98]G. Cugola, E.D. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of ICSE'98*, pages 261–270, 1998.
- [CRW00]A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of ACM PODC 2000*, pages 219–227, 2000.
- [DEG02]C.H. Damm, P.T. Eugster, and R. Guerraoui. Abstractions for distributed interaction: Guests or relatives? In Progress, 2002.
- [DHTT00a]C.H. Damm, K.M. Hansen, M. Thomsen, and M. Tyrsted. Creative object-oriented modelling: support for intuition, flexibility, and collaboration in CASE tools. In *Proceedings of ECOOP 2000*, pages 27–43, 2000.
- [DHTT00b]C.H. Damm, K.M. Hansen, M. Thomsen, and M. Tyrsted. Tool integration: Experiences and issues in using XMI and component technology. In *Proceedings of TOOLS Europe'2000*, pages 94–107, 2000.
- [EG01]P.T. Eugster and R. Guerraoui. Content-based publish/subscribe with structural reflection. In *Proceedings of Usenix COOTS 2001*, pages 131–146, 2001.
- [EGD01]P.T. Eugster, R. Guerraoui, and C.H. Damm. On objects and events. In *Proceedings of ACM OOPSLA 2001*, pages 131–146, 2001.
- [EGS01]P.T. Eugster, R. Guerraoui, and J. Sventek. Distributed Asynchronous Collections: abstractions for publish/subscribe interaction. In *Proceedings of ECOOP 2000*, pages 131–146, 2001.
- [Eug01]P.T. Eugster. *Type-Based Publish/Subscribe*. PhD thesis, EPFL, Lausanne, Switzerland, 2001.
- [FMMB02]L. Fiege, M. Mezini, G. Mhl, and A.P. Buchmann. Engineering event-based systems with scopes. In *Proceedings of ECOOP 2002*, pages 309–330, 2002.
- [GHJV95]E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Software*. Addison-Wesley, 1995.
- [McL93]M.J. McLennan. [incr Tcl]: object-oriented programming in Tcl/Tk. In *Proceedings of the Tcl/Tk Workshop*, pages 31–38, 1993.
- [OMG01]OMG. Unified Modeling Language specification 1.4. Technical Report formal/01-09-67, Object Management Group, 2001.

- [Par92]P. Pardyak. Group communication in an object-based environment. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems IWOOS'92*, pages 106–116, 1992.
- [Pra99]A. Prakash. Group editors. In M. Beaudouin-Lafon, editor, *Computer Supported Cooperative Work*, pages 103–133. Wiley, 1999.
- [Rei90]S.P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(7):57–66, 1990.
- [Rog97]D. Rogerson. *Inside COM. Microsoft's Component Object Model*. Microsoft Press, 1997.
- [SA97]B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG'97*, 1997.
- [Sar01]S. Sarni. Design and implementation of a type-based publish/subscribe architecture. Technical report, EPFL, Lausanne, Switzerland, 2001.
- [Tal99]Talarian. Everything you need to know about middleware. Technical Report <http://www.talarian.com>, Talarian Corporation, 1999.
- [TIB99]TIBCO. TIB/Rendezvous white paper. Technical Report <http://www.rv.-tibco.com/>, TIBCO Inc., 1999.
- [Wik98]J. Wikman. Evolution of a distributed repository-based architecture. In *Proceedings of the First Nordic Workshop on Software Architecture NOSA '98*, 1998.