Two Case Studies of Subsystem Design for General-Purpose CSCW Software Architectures

Paul Grefen, Klaas Sikkel, Roel Wieringa
Department of Computer Science,
University of Twente, The Netherlands
{grefen,sikkel,roelw}@cs.utwente.nl

Abstract

This paper discusses subsystem design guidelines for the software architecture of general-purpose computer supported cooperative work systems, i.e., systems that are designed to be applicable in various application areas requiring explicit collaboration support. In our opinion, guidelines for subsystem level design are rarely given – most guidelines currently given apply to the programming language level. We extract guidelines from a case study of the redesign and extension of an advanced commercial workflow management system and place them into the context of existing software engineering research. The guidelines are then validated against the design decisions made in the construction of a widely used web-based groupware system. Our approach is based on the well-known distinction between essential (logical) and physical architectures. We show how essential architecture design can be based on a direct mapping of abstract functional concepts as found in general-purpose systems to modules in the essential architecture. The essential architecture is next mapped to a physical architecture by applying software clustering and replication to achieve the required distribution and performance characteristics.

Keywords

Architecture design, subsystem design, case study,
workflow management system, groupware system.

1 Introduction

In situations where information systems are complex, e.g. because of advanced functionality, distribution, or interoperability requirements, subsystem design guidelines are indispensable to arrive at well-structured and maintainable systems. Guidelines for subsystem design of general-purpose software are rare. Structured and object-oriented design methods mostly ignore the subsystem level [You93, Rum91], and those authors who do discuss this level, confine themselves to listing frequently identified subsystems without giving the rationale behind these lists [Awa96, Gom93, Shl92]. None of these authors discuss subsystem design guidelines for general-purpose systems. The design of these systems poses special problems, because we cannot use a concrete application domain to guide system decomposition, as is often recommended by OO methods. This certainly holds for the design of CSCW systems, as these are used in highly varying contexts.

The field of patterns and software architecture mostly focuses on the programming language level [Gam95, Bus96]. In addition, many of these approaches, especially the structured and object-oriented ones, discuss ways to *represent* architectural designs rather than ways to arrive at good designs. This syntactic orientation sometimes appears as a preoccupation with a particular programming language, such as Ada [Gom93, Shu91] or C++.

In this paper, we address the issue of subsystem design of general-purpose CSCW software in two case studies of real-world software development. The first case study is based on the experiences in the WIDE ESPRIT project, in which a large software organization and two academic groups cooperated in software design. We discuss subsystem-level design guidelines that were used in the development of the WIDE prototype workflow management system [Cas96, Cer97, Gre99a]. In this development process, an existing commercial workflow management system has been extended with advanced data support, transaction support, and active rule support. A solid architecture design has been a requirement here to obtain a flexible, distributed, and scalable system. The fact that we deal with a general-purpose system implies that a concrete application domain is not available at design system time – thus requiring an alternative design approach. For reasons of clarity and brevity, we focus on the advanced transaction support subarchitecture [Gre97, Boe98, Gre99b] in this case study. This treatment is an elaboration of the position we have formulated in a previous short position paper [Gre98b]. The second case study is based on the development of a widely used, web-based groupware system [Ben95, Ben96, Ben97]. In this case study, we validate the guidelines extracted from the first case study by applying them to the design of the groupware system. Also, the design of this groupware system is influenced by the fact that a concrete application domain is not available at system design time.

The structure of this paper is as follows. In section 2, we start with summarizing related work in subsystem-level design. In section 3, we present the major design decisions made in the WIDE system and in section 4, we discuss the lessons learned from this project. We especially focus on extensibility of the architecture, because that was one of the design goals of the system. In section 5, we validate the lessons learned in an analysis of the design decisions made for the BSCW group support system. In Section 6, we present our conclusions.

2 Related work

Recently, there has been a vast interest in software architectures, as represented by numerous papers and books on the subject. Well-known work is this respect is for example [Sha96] and [Bas98]. Generally addressed aspects are architectural styles, architectural infrastructures, and architecture specification. Little attention is paid, however, to guidelines for high-level subsystem design in real-world architecture development. In this paper, we focus on this aspect by exploring two cases.

Shlaer and Mellor [Shl92] define subsystems in terms of domains. They define a *domain* as a "separate real, hypothetical, or abstract world inhabited by a distinct set of objects that behave according to the rules and policies characteristic of a domain". They recommend defining a subsystem for each domain of the system and give the following examples of domains: the user interface domain, the subject domain, the database domain and the operating system domain. Elaborating this approach, Awad et al. give additional examples [Awa96]: the fault tolerance domain, the performance monitoring domain, alarm handling, hardware control and the device domain. Although these examples are useful, they do not help clarifying the concept of domain. The concept of a separate world used in the definition of the domain concept is at best metaphorical, and it is hard to see how a set of objects could be anything but distinct. Eliminating vague concepts from the definition, we are left with the concept of a domain as a part of the world for which there are characteristic rules and policies. This still leaves considerable freedom in deciding what is and what is not a domain. Moreover, it is not clear why the examples given are good domains. Defining a subsystem for all of these domains may lead to an unnecessarily complex system architecture.

Gomaa [Gom93] also lists frequently identified subsystems in control-intensive systems, such as subsystems for real-time control, real-time coordination, data collection and data analysis, and servers. Here too, one may recognize the idea of domains, but again it is not clear why one should identify these subsystems and not others.

What is needed in the design of complex systems is not only a list of frequently identified subsystems, but also an understanding of the reasoning behind subsystem design, so that this reasoning can be applied in different cases with possibly different results. This is especially necessary for general-purpose systems, because these may be used in many different contexts, so that it is less apparent what the application domain of the system is. A more general point of view is thus necessary that does not depend on lists of frequently identified subsystems or on the concept of a domain. In this paper, we make a first step into this direction.

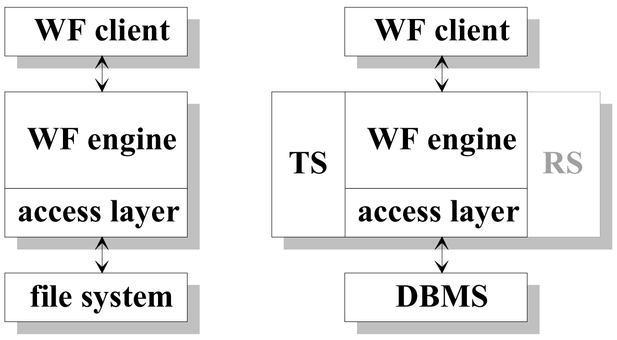

Figure 1 a, b: physical FORO architecture and overall WIDE architecture

Finally, we mention the 4+1 views approach of Kruchten [Kruc95] to architecture design, in which logical view (user's perspective), process view (programmer's perspective), development view (system integrator's perspective), and physical view (system engineer's perspective) are distinguished. The logical view is strongly related to our notion of essential architecture, whereas the physical view is related to our notion of physical architecture. In our opinion, in the design of general-purpose software, the process view and development view have to cater for a 'variable' physical view, as the deployment environment of a system is not known at system design time.

3 Design of the WIDE workflow management system

WIDE (Workflow on Intelligent Distributed database Environment) is an ESPRIT project aiming at complementing standard relational database functionality with advanced transaction management and exception handling to provide an infrastructure for business process support software, such as workflow management systems [Cas96, Cer97, Gre99a]. To reach this aim, advanced transaction and rule support has been added to a commercial relational database management system (DBMS). The WIDE project uses the Oracle DBMS, but other relational systems can be used too. In addition, the commercially available FORO basic workflow management system, built on top of an indexed file system, was reengineered and extended so that it can run on top of a relational DBMS extended with WIDE transaction and rule support functionality. FORO is produced and put into the market by one of the partners in the WIDE project, Sema Group [Sem96].

Figure 1a shows the physical architecture of the basic FORO system, consisting of a workflow engine, a workflow client, and an underlying file system. The workflow engine is shielded from the underlying file system by an access layer that hides specific details of the file system and offers a simple object-oriented interface to the workflow engine. Shadowed rectangles in the figure represent modules that at the implementation level are executables with their own heavyweight process. In the WIDE project, this physical architecture was extended with transaction support (TS) and rule support (RS), and the file system was replaced by a relational DBMS, as depicted in Figure 1b [Cer97, Gre98a]. For reasons of brevity and clarity, we focus on the design of subsystems of the TS extension in this paper.

To get from the physical architecture of Figure 1a to that of Figure 1b, a classical reengineering cycle was followed, as recommended for example by McMenamin and Palmer [McM84]. In this cycle, an essential model of the current physical system has been abstracted, then transformed into an essential model of the desired system, and finally mapped to and implemented in a physically distributed environment. Below, we present a rational reconstruction of this design process.

3.1 Designing the desired essential architecture

We define an *essential architecture* as an architecture that is defined exclusively in terms of the external usage environment of the system and of the role that the system plays in this environment. It may for example define the architecture in terms of external entities or external functionality or in terms of application domain entities. An essential architecture is abstract in the sense that it completely abstracts from the underlying implementation environment. In particular, it abstracts from the distribution of the software over physical resources (allocations of runtime processes to workstations or servers). The essential architecture is sometimes referred to as the *logical* or *conceptual* architecture. The term “essential” was introduced by McMenamin and Palmer [McM84] and expresses the fact that the architecture is invariant under all possible changes of implementation. In our approach, we use the essential architecture of a system as focal point of combining structured and object-oriented decomposition criteria [Wie96, Wie98a].

To obtain an overall essential architecture of Figure 1a, we ignore the fact that the access layer is implemented as a static function library without its “own” process and model it as a separate module. This is our first step in transforming the current physical architecture into an essential architecture. The resulting architecture is shown in Figure 2a. The rectangles in our representation of essential architectures represent conceptually concurrent processes. Viewed from the standpoint of the external requirements of the system, these modules represent processes that occur concurrently and each have a specific functionality.

To transform this architecture into the desired essential architecture, we make two steps. Firstly, we observe that workflow transactions are supported in WIDE by an orthogonal two-layer transaction model [Gre97], consisting of a global transaction layer and a local transaction layer. Global transactions model long-running workflow processes, local transactions relatively short-living workflow subprocesses. Each global transaction consists of a number of local transactions. Each local transaction has a semantics that is independent from the global transaction of which it is a part. Accordingly, we decompose the transaction support extension into two separate subsystems: global transaction support (GTS) and local transaction support (LTS), where only LTS requires access to underlying DBMS transaction facilities. Secondly, we replace the file system by the DBMS. After these steps, we arrive at the essential architecture depicted in Figure 2b. Note that these design decisions are made by referring to the desired functionality of the system and not by referring to the application domain.

Secondly, we further decompose the GTS and LTS subsystems. To do this, we first observe that at run-time, the WF engine contains a number of active workflow process instances (workflow cases), which correspond one-to-one with business process instances. Since there is a dynamically varying set of business process instances, the engine contains a dynamically changing set of workflow process instances. Here, we see a domain-oriented decomposition principle at work.

To decompose the GTS subsystem, we observe that it manages global transactions that have a one-to-one relationship with active workflow process instances. At the start of a process instance, a global transaction is created and at workflow completion, its global transaction is destroyed. The state of a global transaction is influenced by high-level business events that occur in the corresponding process instance. The global transaction state is stored persistently in the DBMS through the access layer. Apart from state administration, logic is required to compute compensating global transactions in case of global abort events [Gre97, Gre99b]. In accordance with this functionality, the GTS subsystem is decomposed into a dynamic set of global transaction (GT) modules that interface with the WF engine (event signaling) and the access layer (persistent storage), and a persistent global transaction manager module (GTM) containing the compensation logic.

To decompose the LTS subsystem, we observe that it manages local transactions that have a many-to-one relationship with active workflow instances [Gre97]. Local transactions are created dynamically at the start of certain tasks in a workflow and disposed of at the completion of these tasks. Their state is influenced by low-level business events in the corresponding business process. Because of their dynamic nature, local transactions are mapped to local transaction (LT) modules. To relieve the WF engine from the one-to-many communication between a workflow instance and its local transactions, a local transaction manager (LTM) is inserted between engine and LTs that performs a dispatching function. This corresponds to the many-to-one pattern of Gamma et al. [Gam95]. As LTs perform transactional operations on the underlying DBMS and abstraction is required with respect to specific DBMS platforms, a local transaction interface (LTI) is inserted between LTs and DBMS. The LTI shields DBMS-specific interface details and manages connections to the DBMS. Experience has turned out that proper handling of connections to the DBMS (pooling of connections) is of great influence on the overall behavior of the system. For transactional operations, the LTI acts like the access layer for data manipulation operations. This structure is similar to the proxy pattern in Buschmann et al. [Bus96].

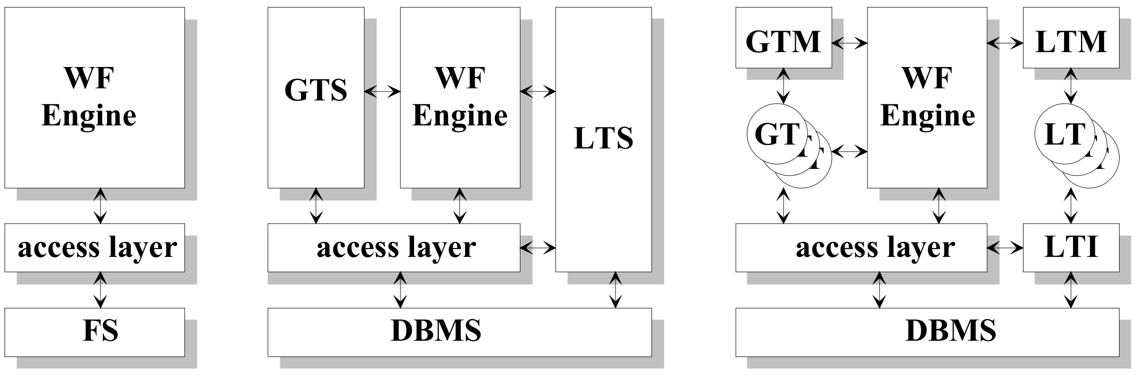

Figure 2 a, b, c: Essential architectures of basic system, desired system, detailed desired system

The result of GTS and LTS decomposition is shown in Figure 2c. In this figure, circles represent objects that are created and disposed of dynamically. This is the desired essential architecture of the extended WFMS. The functions of the modules can be summarized as follows:

Workflow Engine: provides the basic workflow functionality to its workflow clients; generates workflow process events for the transactional modules and receives process status information from them.

Global Transaction Support:

Global Transaction Object (GT): processes high-level workflow process events and stores status information of one instantiation of a global transaction; answers GTM information requests.

Global Transaction Manager (GTM): computes global compensation patterns (rollback processes) for a global transaction upon request of the WF Engine, using status information from the corresponding GT object.

Local Transaction Support:

Local Transaction Object (LT): processes low-level workflow events and stores status information of one instantiation of a local transaction; passes low-level transactional operations to the LTI.

Local Transaction Manager (LTM): handles the communication between the WF Engine and LT objects by dispatching messages.

Local Transaction Interface (LTI): translates abstract low-level transactional operations to specific DBMS operations and manages the physical connections to the DBMS.

Access Layer: translates object-oriented data access operations to specific DBMS relational data manipulation operations.

In the essential architecture, there are many GT and LT objects but of the other modules, there is exactly one instance each.

3.2 The desired physical architecture

To go from essential to physical architecture, we have to take software clustering into heavyweight processes and allocation of these processes to physical resources into consideration.

We first observe that each GT instance and each LT instance is related to one workflow instance. Communication between a GT object and its workflow instance in the WF engine is frequent. The same observation holds for LTs. Hence, it is efficient to place GT and LT instances in the same process that holds the corresponding workflow instance.

Next, we observe that LTM functionality is relatively simple and that communication between a workflow instance and the LTM is frequent, so it is efficient to place the LTM in the same process as the WF instance is placed. Functionality of the GTM, on the other hand, is relatively complex [Gre99b] and communication between GTM and other modules is infrequent, so there is no reason to

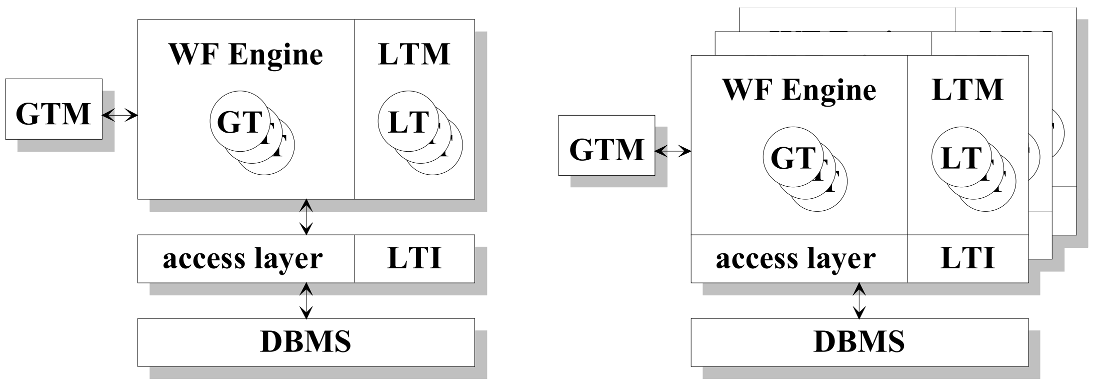

Figure 3 a, b: Intermediate physical architecture, final physical architecture

combine the GTM with other modules. This means that workflow engine, LTM, and GT/LT modules are all clustered into one process, but that GTM is kept as a separate process. As the access layer and the LTI perform similar and relatively simple functions, they are clustered too. The result is shown in Figure 3a. Each shadowed rectangle in this figure represents an executable with its own heavyweight processes, which may contain multiple threads. Each rectangle within a shaded rectangle represents a logical task to be accomplished by the heavyweight process, that may be implemented as a separate thread. At this level, though, this has not yet been decided.

Next, clustering of the combined access layer and workflow engine modules is considered. Given the observation that the state of both extended access layer and extended workflow engine can be partitioned on a per-workflow-instance basis, clustering is possible. Given very frequent communication between engine and access layer, clustering is desired. This results in the physical clustering shown in Figure 3b.

The WF engine and GTM clusters can be replicated arbitrarily on different processors, to allow for a flexible coupling of workflow engines to GTM servers. To do this in a flexible and location-transparent way, middleware functionality is required. In WIDE, a CORBA object request broker [OMG95, Sie96] is used. This allows for a flexible instantiation of modules at runtime with transparent allocation. This means for example that communication between WF Engine and GTM is location-transparent. Consequently, one-to-many or many-to-one coupling between these modules possible, depending on load characteristics. Finally, to allow for flexible coupling of a workflow engine to different DBMS servers (which each may serve multiple workflow engines), a client/server coupling has been used [McC96].

4 Discussion of the WIDE case study

The WIDE system has a number of characteristics that are relevant for the design of its software architecture as discussed in the introduction:

- It is strongly related to highly complex business processes. This implies a need for an *extensible* architecture with respect to support for advanced features like extended transaction management.
- It is a *general-purpose* system to be parameterized for specific workflow applications. This implies a need for a high-level and flexible approach to module replication and allocation, such that throughput and response times can be tuned to load characteristics.
- It is *distributed*, as dictated by the inherently distributed nature of workflow applications. This implies a need for a well-structured design into distributed physical modules.
- It has a soft real-time character related to supporting business processes. This means that special care must be taken in the design of the communication structures such that *performance* is guaranteed.

We now discuss how these characteristics are reflected in guidelines used in the design process.

Extensibility. By distinguishing the desired essential architecture from the desired (and realized) physical architecture, we create the freedom to map essential subsystems in various ways to physical resources, without changing the essential model. In the design process outlines above, we have shown how this mapping can be performed in a structured way to achieve desired distribution and performance characteristics. The essential model as the centerpiece of the design thus improves the traceability of the resulting software to essential subsystems and ultimately to requirements. This will highly contribute to extensibility and maintainability of the software: traceability of software to requirements makes it possible to easily locate affected subsystems in case of extension or modification. In addition, by using functional decomposition as the essential decomposition guideline, extensibility of functionality is enhanced: adding new functional concepts to a system will easily translate to changes to the essential architecture and from there to the physical architecture.

Another ingredient in achieving flexibility used in the WIDE system (only discussed briefly in this paper) is middleware like the object request broker infrastructure. This helps realizing a clear separation between essential and physical architecture without sacrificing flexible distribution or performance, and contributes to location-independence.

General-purpose system. In the design of the WIDE system, *desired external functionality on the conceptual level* has been used as a criterion for decomposing the system at the essential subsystem level (for example, the conceptual transaction model). The reason is simple: for general-purpose systems, there is no specific application domain in terms of which we can partition the system. The object-oriented criterion of using the application domain as decomposition principle therefore is not applicable. On the other hand, there is a clear idea about the desired functionality of the system. We think functional decomposition in general is a sound high-level design principle for general-purpose systems.

Distribution. In agreement with the usual distinction between essential and physical design, we defer all decisions about the distribution of components to the physical design level. At the essential level, we decompose the system in terms of criteria taken from the external usage environment and have complete disregard for implementation matters. So we do not refer to the distribution architecture of the underlying implementation platform or properties of the programming language. Like [Awa96], at this level of abstraction, we assume unlimited concurrency of the components.

During physical design, we take communication properties into account to decide about clustering and allocation of components to resources (processors). This may result in a many-many relationship between modules at the essential and physical levels.

Performance. Decisions about clustering and allocation of modules to resources were made according to the frequency of communication between components and the need for the ability to replicate certain functionality at different locations (to match load characteristics). This optimizes performance for a chosen essential architecture. Again, the use of middleware services provides the necessary mechanisms to easily use module replication.

Returning to our discussion in Section 2, we tried to go beyond merely giving lists of frequently encountered subsystems by giving the reasoning behind our partitioning into subsystems. The top-level essential decomposition is that the added functionality to a relational DBMS should be advanced transaction and exception support. Next, we distinguish global from local transactions, etc. As a result, the summary of module functionality given earlier reads like a function refinement tree of the required advanced transaction support.

We should add that functional decomposition at the subsystem level is certainly not the only guideline that can be used, not even in general-purpose systems (where the specific application domain is not known in advance). Other criteria that refer to the external environment can also be considered. For example, in control-intensive systems, one may define subsystems for particular classes of external devices or for particular classes of events that share periodicity properties [Shu92, Gom93, You93]. Functional decomposition will play a central role in these cases at the subsystem software design level.

With respect to the issue which comes first, the partitioning into sequential (mutually concurrent) processes or the partitioning into objects, we also agree with Shumate [Shu91]. The design of software objects comes into sight only after modules have been allocated to resources (processors), so that we know that the software objects in one physical module will be part of a heavy-weight sequential process. Of course, a first object-oriented design of a physical module may ignore this, but eventually, the design has to allocate all these objects to heavyweight processes. The designer may then decide to allocate some objects their own thread, but we have seen in Figure 3b that here too, some large-grained objects may be identified in a functional way.

We conjecture that the guidelines extracted from the WIDE project are generally applicable to the design of extensible, general-purpose, distributed software systems. In the next section, we therefore briefly look at the design principles used in a second general-purpose, distributed system.

5 Application in a second case study: BSCW

Basic Support for Cooperative Work (BSCW) [Ben97] is a groupware system offering shared workspaces accessible via the World Wide Web. A shared workspace is a repository for shared information, which may contain different kinds of objects and provides some meta-information about these objects to members of the workspace. For more information about the system, see its Web site (`bscw.gmd.de`) or [Ben97].

BSCW was developed at the Institute for Applied Information Technology of GMD (German Research Center for Information Technology). GMD-FIT runs a public BSCW server with some 11,000 registered users and in 1998 a daily average of 15,700 accesses. Between August 1996 and December 1998 13,000 copies of the BSCW server software have been downloaded from GMD's web site. The BSCW software runs on common variants of UNIX and on Windows NT. BSCW is a general-purpose system in the sense that performance characteristics and requirements are not known. Shared workspaces support a variety of different kinds of activities and a server may serve any number of users groups. Server load may vary from virtually nothing for many local installations to peak loads of over 5,000 requests per hour at the public BSCW server at GMD.

An important aspect in the design of the system was to minimize requirements for the system and software needed by its users. End users access the system with an ordinary Web browser. The system's user interface is in essence handled at the server side and communicates with the user through conventional Web technology. The BSCW server is realized as an auxiliary component to some (arbitrary) Web server. Here we consider Version 2 [Ben96, Ben97], which is a full reimplementation of Version 1 [Ben95] with a modified architecture. To our opinion, architectural analysis as presented in this paper is important in the context of evolution of software to keep track of high-level design issues that might else be reflected in program code only.

BSCW is a distributed system, but in order to avoid the need for special client software at the end user's side the distribution relies on common Web technology. Hence the system's implementation comprises only the server software. BSCW is smaller and simpler than WIDE, but also in this system we find functional decomposition and unlimited concurrency at the essential architecture level. And just as in the WIDE system, the physical architecture consists of communicating sequential processes the structure of which is motivated by performance requirements.

The essential architecture of BSCW is shown in Figure 4. Activity occurs in the form of requests, which may affect objects stored at the server and always are followed by a response. For an incoming request a request object is generated by a request/response translator. In the design it was anticipated that a BSCW server may have interfaces for several interaction protocols (but only the WWW interface has been realized). The request/response dispatcher passes the request object to an appropriate operation handler, which modifies and/or retrieves data from the object store. A response object is returned, from which the translator generates an HTML page. For each type of operation there is a separate operation handler, allowing for easy extension with new operations in future. This corresponds to the concept of event partitioning introduced by McMenamin and Palmer [McM84].

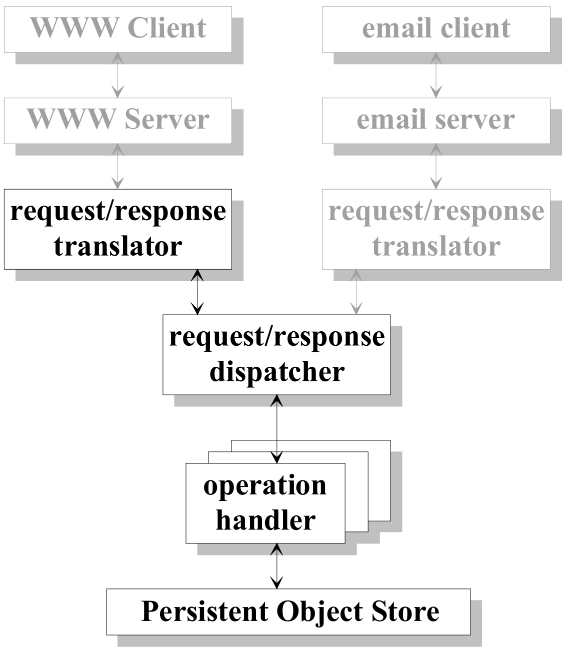

Figure 4: Essential architecture of BSCW

The essential architecture focuses on separation of functionality in logical modules – as we have shown with the WIDE essential architecture. At the essential level, important implementation aspects like how to handle multiple requests are abstracted away. Figure 5 shows the physical architecture of the system.

In Figure 5a, the components of the essential architecture are mapped onto physical processes. Request handling and operation handling are collapsed into a single component, comprising modules for request/response translation and for the operation handlers. This is motivated by reasons of efficiency (avoiding unnecessary inter-process communication) and cardinality (a single instance of each module is needed for handling a single request). These motivations parallel those for the clustering in the physical design of the WIDE system. For each request a separate process is instantiated (the CGI process [Bas98] invoked by the WWW server); multiple instances can exist simultaneously. A separate component is needed to handle persistent object storage. The CGI processes execute queries on the object store by means of remote procedure calls.

In order to economize interaction with the persistent object store, the CGI processes obtain a local copy of the needed objects, see Figure 5b. Updates on these local copies are committed to the object store when the operation has been completed (using optimistic concurrency control, the commit fails if meanwhile the object has been modified by another process). A second advantage is that no lower level rollback is needed when for whatever reason a composite operation fails.

The BSCW system is a groupware server that is available worldwide to an unknown future number of users who use the system in unpredictable ways. Given this nature of the system, the request load for a server cannot be predicted and may very much fluctuate in time. Hence it makes sense to dynamically allocate processes according to the number of requests. When the load increases, a foreseeable bottleneck (other than general degradation of the host's performance) is the interface to the permanent object store, which, for obvious reasons, is handled by a single process. In order to alleviate this, some care has been taken to optimize interaction with the data store and the dynamically allocated processes have been equipped with a local object space.

As in the WIDE system, we observe a many-to-many relationship between modules in the essential and physical architecture. Where the essential architecture is motivated by functional requirements, the physical architecture is motivated by performance requirements. At the essential level, we have unrestricted concurrency; at the physical level, we have clustered and allocated modules to physical resources. The essential architecture helps to make functional requirements explicit, provide a logical view of the system that remains invariant under changes of implementation, and trace functional requirements to the physical design.

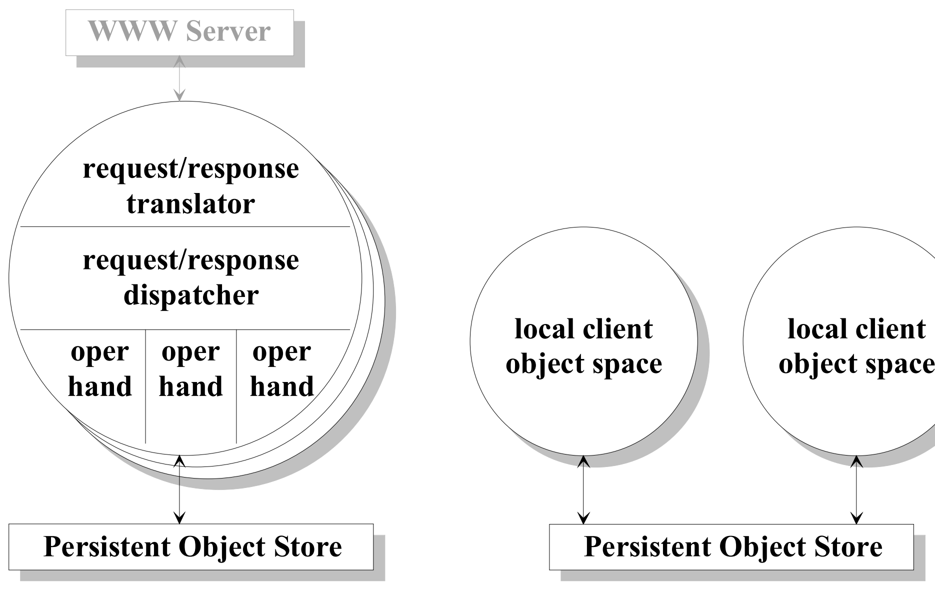
WWW Server
request/response translator
request/response dispatcher
oper hand
oper hand
oper hand
local client object space
local client object space
Persistent Object Store
Persistent Object Store

Figure 5 a, b: Physical architecture of BSCW, BSCW object storage management

6 Conclusion

In this paper, we analyzed the design of architectures of the WIDE workflow management system and of the BSCW groupware system as cases of well-structured general-purpose CSCW systems. We have illustrated that the following design guidelines are important in realizing well-structured general-purpose system architectures:

1. The essential architecture is the centerpoint of system design. In building new systems, the essential architecture is the starting point of design. In rebuilding or extending existing systems, the essential architecture should be abstracted from the existing physical architecture.
2. The essential architecture of a general-purpose system is based primarily on the functional requirements of the system. Instead of a particular application domain, there is an intended usage of these systems, that provides general, abstract concepts like workflow case (WIDE) or shared workspace (BSCW). These concepts are used to define the essential architecture. This may involve the dynamic creation of essential components, such as local and global transactions (WIDE), or requests (BSCW) to match the way of working of the user.
3. There may be a many-to-many relationship between modules at the essential and the physical level. The physical architecture is motivated by performance issues. Most important of these in distributed systems is the minimization of communication overhead. The distinction between essential and physical architecture helps separating the design concerns of flexibility and performance. We propose ensuring flexibility by maintaining a correspondence between functional requirements and essential architecture, and to ensure good performance by minimizing communication overhead. The use of middleware also allows achieving a balance between flexibility (location-independence) and performance.

We conclude with the statement that at the subsystem design level, the opposition between functional and object-oriented design is not very productive. It is argued elsewhere that the essential difference between functional and object-oriented design is the separation of data processing, data storage and control in data flow models versus the encapsulation of all of these in object-oriented models [Wie98b]. Our analysis has shown that at the subsystem design level, this opposition does not exist. Rather, in both approaches we have subsystems and modules that may have a state and external interfaces, and that should have clearly defined functions. The difficult design problem is how to decide which subsystems there are. This problem should not be compounded by an unproductive restriction to design heuristics from only one school of thought.

Acknowledgements

All members of the WIDE team are acknowledged for their parts in the development of the WIDE architecture described in this paper. Specific thanks go to Gabriel Sánchez, Stefano Ceri, Jochem Vonk, and Erik Boertjes. The members of the BSCW team are acknowledged for their parts in the development of the BSCW architecture presented in this paper.

References

[Awa96] M. Awad, J. Kuusela, J.Ziegler, *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion*, Prentice-Hall, 1996.

[Bas98] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.

[Ben95] R. Bentley, T. Horstmann, K. Sikkel, J. Trevor. "Supporting Collaborative Information Sharing with the World-Wide Web: The BSCW Shared Workspace System". *Procs. 4th Int. WWW Conference*. Boston, USA, 1996, pp. 63-73.

[Ben96] R. Bentley, U. Busbach, K. Sikkel. "The Architecture of the BSCW Shared Workspace System". *Procs. ERCIM/W4G Workshop on CSCW and the Web*. Arbeits-papiere der GMD 984 (1996), pp. 31-41.

[Ben97] R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, K. Sikkel, J. Trevor, G. Woetzel. "Basic Support for Cooperative Work on the World Wide Web". *Int. Journal on Human-Computer Studies*. Vol. 46, 1997, pp. 827-846.

[Boe98] E. Boertjes, P. Grefen, J. Vonk, P. Apers; "An Architecture for Nested Transaction Support on Standard Database Systems". *Procs. 9th Int. Conf. on Database and Expert System Applications*. Vienna, Austria, 1998.

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *A System of Patters – Pattern-Oriented Software Architecture,* Wiley, 1996.

[Cas96] F. Casati, P. Grefen, B. Pernici, G. Pozzi, G. Sánchez**,** *WIDE: Workflow Model and Architecture*, CTIT Technical Report 96-19, University of Twente, 1996.

[Cer97] S. Ceri, P. Grefen, G. Sánchez, "WIDE - A Distributed Architecture for Workflow Management". *Procs. 7th Int. Workshop on Research Issues in Data Engineering,* Birmingham, UK, 1997, IEEE, pages 76-79**.**

[Coo94] S. Cook, J. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice-Hall, 1994.

[Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns—Elements of reusable Object-Oriented Software,* Addison-Wesley, 1995.

[Gom93] H. Gomaa, *Software Design Methods for Concurrent and Real-Time Systems*, Addison-Wesley, 1993.

[Gre97] P. Grefen, J. Vonk, E. Boertjes, P. Apers, "Two-Layer Transaction Management for Workflow Management Applications", *Procs. 8th Int. Conference on Database and Expert System Applications*, Toulouse, France, 1997, Springer LNCS, pages 430-439.

[Gre98a] P. Grefen, S. Ceri, G. Sánchez, *Transaction and Rule Support for Workflow Management - A Retrospective on the WIDE Architecture*, CTIT Technical Report 98-16, University of Twente, 1998.

[Gre98b] P. Grefen, R. Wieringa, "Subsystem Design Guidelines for Extensible General-Purpose Software", *Procs. 3rd Int. Software Architecture Workshop*, Orlando, USA, 1998.

[Gre99a] P. Grefen, B. Pernici, G. Sánchez (Eds.), *Database Support for Workflow Management: The WIDE Project*, Kluwer Academic Publishers, 1999.

[Gre99b] P. Grefen, J. Vonk, E. Boertjes, P. Apers, "Semantics and Architecture of Global Transaction Support in Workflow Environments", *Procs. 4th IFCIS Conf. on Cooperative Information Systems*, Edinburgh, Scotland, 1999; pp. 348-359.

[Kruc95] P.B. Kruchten, "The 4+1 View Model of Architecture", *IEEE Software*, November 1995:42-50.

[McC96] D. McClanahan, *Oracle Developer's Guide*, Osborne McGraw-Hill, USA, 1996.

[McM84] S.M. McMenamin, J.F. Palmer, *Essential Systems Analysis,* Prentice-Hall, 1984.

[OMG95] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Version 2.0*, Object Management Group, 1995, http://www.omg.org.

[Rum91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design,* Prentice-Hall, 1991.

[Sem96] FORO Manual Set; Sema Group sae, Madrid, Spain, 1996.

[Sie96] J. Siegel; *CORBA Fundamentals and Programming*; Wiley & Sons; New York, USA, 1996.

[Sha96] M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[Shl92] S. Shlaer, S.J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice-Hall, 1992.

[Shu91] K. Shumate, "Structured analysis and object-oriented design are compatible", *Ada Letters*, 11(4):78-90, May/June 1991.

[Shu92] K. Shumate, M. Keller, *Software Specification and Design: A Disciplined Approach for Real-Time Systems*, Wiley, 1992.

[Wie96] R.J. Wieringa, *Requirements Engineering: Frameworks for Understanding*, Wiley, 1996.

[Wie98a] R.J. Wieringa, "Postmodern software design with NYAM: Not Yet Another Method", *Procs. NATO Workshop on Requirements Targeting Software and Systems*, Springer LNCS 1526:69-94, 1998.

[Wie98b] R.J. Wieringa, "A survey of structured and object-oriented software specification methods and techniques", *ACM Computing Surveys*, 30(4):459-527, December 1998.

[You93] Yourdon Inc, *The Yourdon(TM) Systems Method: Model-Driven Systems Development*, Prentice-Hall, 1993.