Dr. Ole J. Anfindsen, founder
Xymphonic Systems AS
Oslo, Norway
olejoan@online.no

Revision 3: 15 October 2008
Revision 2: 2 October 2003
Revision 1: 28 June 2002

The Power of Xymphonic™ Collaboration

Executive summary

The purpose of this paper is to discuss a particular style of collaboration referred to as *xymphonic*[1]. After a presentation of the general challenges involved in supporting collaborative work, a user-oriented (i.e., non-technical) presentation of the xymphonic model of collaboration is given. The xymphonic model is compared to alternative ways of supporting collaboration, and shown to be superior.

1 Introduction

Collaboration between human beings is important in any civilized society. This paper discusses how computers can help people collaborate. Carefully note that there are several kinds of collaboration, and that only a particular kind will be discussed here; one which will be referred to as *data-centric* collaboration. Data-centric collaboration is characterized by two or more people working concurrently on a *shared pool of data resources*. Moreover, such work should not be considered collaborative, unless the people involved are working towards a common goal. An example of data-centric collaboration would be a group of people working simultaneously on a large document. Another example would be a group of engineers working together on the design of, say, an automobile, an airplane, or a ship.

We shall shortly have a look at the challenges involved in providing good support for data-centric collaboration, and the pros and cons of various possible approaches to those challenges. But first some additional introductory material is presented.

---

[1] Xymphonic™ and Xymphony™ are trademarks of Xymphonic Systems AS.

2 Background

This section is based on material from (Anfindsen 2002), but is included here to provide context for new readers. Those who are familiar with the just mentioned paper, or similar ones by the author, may skip directly to Section 5.

The technology discussed in this paper has its roots firmly planted in the database world, and, more specifically, in transaction theory. Sadly enough, transactions are often poorly understood by computer professionals, sometimes even by database people. However, the transaction model describes the *behavior* of the database as a whole, and is actually one of the two main components of the foundation of database management systems (DBMSs). The other main component is the data model, describing the *structure* of stored data (the most popular one is now the relational model of data).

Xymphonic collaboration is a particular style of collaboration based on the xymphonic transaction model, or simply the xymphonic model. This model is a generalization of the classical transaction model that (to a large degree, at least) has been used in all state of the art DBMSs for decades. The xymphonic model by default supports exactly the same behavior patterns as the classical model, but allows users to deviate from classical behavior in order to support collaboration. While the classical model is perfect for many application domains, it precludes collaboration and is therefore counterproductive for collaborative applications.

Thus, the xymphonic model supersedes the classical one. Xymphonic transactions provide the good, old ACID-properties (atomicity, consistency, isolation, durability) to all applications for which this is the optimal mode of execution, while at the same time allowing other applications to explore the new world of *xymphonic collaboration*.

The importance of collaboration, and the business opportunities associated with good solutions in this area, are discussed e.g. in (Anfindsen 2002), so nothing much will be said about those topics here. The xymphonic model is defined, described, and discussed by Anfindsen (1997), and available in electronic form from the author. Even so, a few words about the model are included below.

3 Two simple extensions to the classical model

This section is based on material from (Anfindsen 2002), but is included here to provide context for new readers. Those who are familiar with the just mentioned paper, or similar ones by the author, may skip directly to Section 5.

The xymphonic model has the classical model as a default, but adds two new and powerful features that enable users to work *xymphonically* whenever they need to.

The most central concept in the xymphonic model is that of a *xymphony,* which is a dynamically created *subdatabase* or *nested database.* By adding recursion and a few other things to standard concurrency control algorithms, a whole new set of possibilities opens up, and arbitrarily complex patterns of interaction between two or more users can be established. Using xymphonies just to deal with simple sharing of data between readers and writers would however be quite awkward. The xymphonic model therefore offers a much more convenient mechanism for this, based on user-defined *access parameters*. This enables users to selectively share data with one another, and also serves as a vehicle for providing meta-information about the data objects in question (such as e.g. reliability, maturity, or degree of completeness).

Taken together, xymphonies and access parameters enable *any* potential conflict between transactions to be handled in a simple yet controlled fashion. The formal foundation for this consists of two simple generalizations of classical *serializability* theory.

4 Simplicity and power: some technical background

This section is based on material from (Anfindsen 2002), but is included here to provide context for new readers. Those who are familiar with the just mentioned paper, or similar ones by the author, may skip directly to Section 5.

The xymphonic model is aimed at bringing the familiar benefits of transaction management to application domains that need long-lasting or collaborative transactions. It is generally acknowledged that so-called classical transactions are unsuitable for this purpose (see e.g. Gray & Reuter 1993, page 180). The problem of collaborative work is therefore well known in computer science, and lots of research papers have been published on this topic. A few examples include (Alonso et al 1995; Kaiser 1990, 1995; Korth & Speegle 1990, 1994; Kumar & Wong 1993; Nodine & Zdonic 1992; Rauft et al 1990), and many additional references are given in (Anfindsen 1997).

While a considerable number of new transaction models have been proposed in the research literature over the years in order to deal with these problems, and although these models have many theoretically interesting features, they tend to be too complicated to implement and use. Thus, they have had very little commercial impact. Xymphonic transactions are radically different from these other transaction models, and to the best of our knowledge, no other model is even close to having the xymphonic model's combination of *simplicity and power*. This unique combination of simplicity and power, combined with an obvious need for collaborative transactions in several application domains (see below), is what convinces us that xymphonic transactions have an important role to play in the years ahead.

The simplicity of the xymphonic model is evident at four levels:

- It has a simple formal foundation based on a straightforward mathematical generalization of classical transaction theory.
- It can be implemented by relatively modest extensions to the data structures and algorithms used in lock managers and similar transaction-related components.
- Application programmers can easily exploit its functionality.
- It enables end-users to navigate the *xymphonic space* simply by clicking on menu options, buttons, check boxes, and other familiar GUI-elements found in all state-of- the-art desktop applications.

Claim 1 is obvious from the definition of xymphonic transactions (Anfindsen 1997), and has repeatedly been pointed out by several reviewers of earlier papers (Anfindsen 1995, 1996abc, 1998ab). Claims 2, 3, 4 have been demonstrated by the implementations of the model.

5 How the xymphonic model supports collaborative work

The challenges of data-centric collaboration show up whenever two or more users need to work with the same data objects *during an extended period of time* (anything from minutes to months). The classical textbook example is that of designers or engineers using a CAD/CAM/CAE tool. On the one hand, they need *access*; both read and write access in the

general case, to each other's portions of the data. On the other hand, they need *control* over their data. And there is an inherent conflict between concurrent access and control.

Thus, when performing data-centric collaboration, one must have some sort of *concurrency control*, or else one will experience so-called *anomalies* like e.g. lost updates or conflicting updates. On the other hand, if there is too much control, people will be prevented from doing what they have to do. This is the dilemma facing anyone who wants to take on the challenges of collaborative work.

In the following, we will illustrate how the xymphonic model deals with these challenges. In Section 7, we will compare xymphonic collaboration with what we believe are the most frequently used alternatives.

5.1 Sharing information with others

People who collaborate usually also need to communicate. If people are co-workers in a xymphony, they probably need to look at other people's contribution every now and then. The xymphonic model makes it very easy to share information, provided the involved parties agree that it should happen.

First of all, it is easy to set up a xymphony such that all participants see the contributions of others as work is progressing. Even so, each participant is working on a local copy of the data on his or her client machine, and can control how often work should be shared with others in the xymphony.

Secondly, unless it has been disallowed by the xymphony creator or system administrator, a participant can choose to place exclusive locks on his or her work, preventing all others from looking at it until it is committed to the xymphony. Carefully note, by the way, that a xymphony can be configured in such a way that no participant can commit work to the xymphony without the approval of the owner or a deputy.

Thirdly, the xymphonic model allows a schema to be created with an arbitrary number of so-called access parameters. By associating such parameters with their data objects writers can inform potential readers of the quality, maturity, reliability, or degree of completeness of their work. Some simple examples of such parameters could be:

- Incomplete draft
- Complete draft
- Approved draft.

More complicated parameter sets can of course be created. Not only could the number of parameters be increased, one could also come up with different *groups* of parameters, having to do with different aspects of the work. For example, there could be multiple parameters for each of the following aspects:

- Managerial
- Legal
- Financial
- Technical

- Marketing
- Security related
- Policy related

And so on. The kind of information one might want to represent for each of the above, could be pending approvals, compliance with corporate policy, quality assurance, and the like. Default schemas could be created for particular industries, upon which could be added further elaborations for individual companies, departments, and projects. Some companies would be happy to use the default schema supplied with their xymphonic product upon installation, others would want to take advantage of the rich customization facilities.

5.2 Compatibility between readers and writers

In the previous section we focused on the way writers use access parameters as a means of communicating the current status of their data to potential readers. However, readers use access parameters too; as a means of being selective as to what data resources to read. Access parameters associated with a read request inform the system of the reader's willingness to read data with a certain quality, reliability, maturity or degree of completeness.

This means that one can e.g. scan for all documents that have reached the level of, say, complete draft, or all documents that are currently being worked on by the legal department, etc. More advanced queries are also possible, see below.

Access parameters are precisely what enable the xymphonic model to support *conditional* compatibility between readers and writers. The default compatibility test is quite straightforward: if and only if write parameters make up a subset of read parameters, are the two compatible (other conditions for compatibility than the just mentioned subset relation are conceivable, but the default seems to cover most of the interesting application requirements).

Aside. Fortunately, one can quite easily and efficiently implement support for conditional compatibility between readers and writers. Parameters can be represented by bitmaps, and one then needs to perform bit-wise AND and OR operations on those bitmaps. **End of aside.**

5.3 This is much better than just allowing dirty reads

Simply supporting dirty reads is not very difficult, and DBMS vendors and others have done so for decades. With dirty reads anyone can read anything anytime. That means dirty reads solve the problem of conflicts between readers and writers, doesn't it? Hardly!

There are at least two fundamental and serious problems with dirty reads:

- Readers have no control over what kind of data they get. In other words, they cannot be selective with respect to the status of the data they read.
- Readers have, in the general case, no way of telling whether a given data item contains committed, stable, and therefore trustworthy information, or something entirely different. In other words, readers are left in the dark as to the status of data they retrieve.

Thus, one could say that dirty read is a do-it-yourself approach to sharing information between readers and writers (but carefully note that it has absolutely nothing to offer when it comes to coordinating multiple writers). If you know what you are doing and you have

sufficient information about the state of the entire system at the time of your query, dirty read might meet your needs, otherwise it will obviously be quite unsatisfactory, and could potentially lead to serious consequences. It is, therefore, well known in the IT industry that dirty read is inadequate in many circumstances, sometimes even counterproductive. In contrast, access parameters, as defined in the xymphonic model, bring *discipline* and *predictability* to the way readers and writers interact.

If a xymphonic system has a mechanism for determining which users are allowed to use which portions of the parameter domain, another interesting possibility emerges. This gives writers a simple tool for selectively sharing their data with some users but not with others. In other words, access parameters can provide a dynamic and fine-grained tool for access control, complementing access control mechanisms at the level of e.g. operating system, application, database, and xymphony.

5.4 Queries on access parameters

In its simplest form, read parameters simply cause the system to skip data items that have incompatible write parameters. But this means that the lock manager component acts as an extension of the query mechanism. Why not take this one step further and allow queries to express conditions having to do with access parameters? And indeed, that is entirely possible. By adding at least one truth-valued function to the query language, one could support quite complicated queries. Here are two examples of the kind of queries that would be possible:

- Give me all design elements that have been approved by the project manager but where approval from the marketing department is pending.
- Give me all documents for which the legal department's approval is pending, and that are currently being worked on either by engineers or sales management.

Aside: One could go even further, and create a single, unified framework for dealing with the following kinds of problems:

- Information is currently unreliable (i.e., subject to change) because someone is working on the data in question.
- Information is unreliable, perhaps permanently so, because the values stored in the system are based on less than perfect input (there could be many reasons why).
- Information is missing (there could be many reasons why).

Such a framework is described in Chapter 7 of (Anfindsen 1997). It includes a generalized version of Boolean algebra, capable of dealing with predicate evaluations in the presence of all sorts of missing and unreliable information. **End of aside.**

6 Example scenario

The following sections describe a scenario that is meant to illustrate how xymphonic collaboration can be used in practice. The examples below are simple and with few people involved. Still, the points thus illustrated would be valid even if the numbers of people, xymphonies, and data resources were much bigger, and even if the application domain were much more complicated than simple editing of text documents (an example of such a complex application domain, which needs better support for collaboration, is CAD/CAM/CAE). Please feel free to extrapolate from the humble scenario below, to some complicated real-world

situation. The tougher the challenges involved, the more the xymphonic model's elegance and power become apparent.

6.1 Brief description of the scenario

As mentioned above, collaboration is important in many application domains. One that is familiar to all of us is writing documents using a word processor, and our example will be drawn from this domain. Small documents usually have a single author only, and so collaboration is not an issue. Larger documents, or documents which contents are complicated, typically need contributions from multiple people.

Let us assume that some company is getting ready to bid for a major contract. Based on specifications from the potential customer, one quickly outlines the contents of the bid document. Alice is the appointed editor and project manager, and she therefore has the responsibility of ensuring that the bid document is ready on time. Below is a simplified scenario describing some of the challenges Alice and her co-workers are going to face.

6.2 Dividing up the work

The company has offices in Bangalore, Oslo, and Boston, and contributions are needed from each of these. Alice therefore creates a skeleton document, with preliminary headers for some of the chapters and subchapters. Now she uses a utility function in her xymphonic collaboration tool (which has been integrated with her word processor) to automatically split the document into a suitable number of subdocuments, then she creates three xymphonies, one for each branch office. In this process, she is prompted to specify which subdocuments to delegate to each xymphony, and who to invite as participants in each xymphony.

6.3 Xymphonies in a distributed environment

Alice created one xymphony per branch office in Bangalore, Oslo, and Boston. Carefully note that a xymphonic collaboration system will allow xymphonies to be distributed. Let's say Alice is located in Boston. Her local server in Boston will then be controlling the entire xymphonic project that Alice is in charge of, but the xymphonies that she created for her colleagues in Bangalore and Oslo, respectively, need not reside on that same server. Once a xymphony is created, the model allows it to be checked out to another machine, and the system ensures that sooner or later the xymphony is checked back in (the latter actually means either committing or aborting the xymphony, because a xymphony is in fact a transaction, albeit a *passive* transaction).

There are some obvious advantages to the model's flexibility with respect to distribution. One is that the server where the xymphonic project originates is eliminated as a single point of failure. That is, even if the server in Boston is down for a while, the xymphonies in Oslo and Bangalore can continue without interruption. And once automatic recovery has been carried out in Boston, the server there will be ready to receive the results from Oslo and Bangalore when those xymphonies are ready to be committed. In fact, the teams in Oslo and Bangalore may never realize the server in Boston was down at all.

A second advantage is that you get better response times when people can work locally. Rather than having to communicate with a server over a global network, collaborators can enjoy working in the controlled environment of a local area network, which usually provides larger bandwidth. Only when xymphonies are created, committed, or aborted, when information is exchanged between xymphonies, or when new resources are added to xymphonies, is there a need for communication between the various geographical sites.

Yet another advantage, which follows from the above, is that the xymphonic model facilitates load balancing between multiple servers by allowing xymphonies to be created and distributed as needed.

6.4 Sharing information within a xymphony

Co-workers in a xymphony are of course able to share information among themselves. There are two basic ways of doing this. One solution is simply to let everyone see all data resources at any time. Another solution is using access parameters to limit who gets access to what at a given point in time.

Carefully notice that each collaborator can control when his/her contributions are made available in the xymphony (what really happens during such within-xymphony publishing, is that data resources are copied from the client machine to the server). This could be done manually or automatically.

Consider for example Arne and Berit working in the Oslo xymphony. Let us assume Arne has some documents marked with parameter "complete draft", and some others marked with "incomplete draft". If Berit *prefers* to read only documents that are complete draft or better, or, alternatively, the xymphony is configured such that she is not *allowed* to see incomplete drafts, an attempt from Berit to read Arne's documents will give access only to some of them.

Aside. This feature of the xymphonic model appears to be more relevant to complicated applications that deal with large numbers of objects in each xymphony, than to simple situations like the ones that the current example will tend to give rise to. For example, the leader of a design project who wishes to assess the progress of his/her team may want to run a query over the contents of a xymphony, filtering away all immature design structures. **End of aside.**

It is worth pointing out that at any point during the life of a xymphony (and its subxymphonies) one can ask the system to compile a snapshot view of everything that is being worked on, or some subset of that which is being worked on. The system knows the mutual relationships of the various data structures, and can automatically put them together to form a complete document (or whatever else the xymphonic participants are about to produce). This is useful e.g. for a project leader who needs to assess progress. Doing so would of course require the necessary access rights.

6.5 Sharing information among multiple xymphonies

The xymphonic model also allows people working in different xymphonies to share information with each other. There are multiple ways in which a xymphonic system could be configured to support this kind of interaction among xymphonies. A simple approach would be to specify for each xymphony which data resources to *export* and which data resources to *import*. Carefully note that resources can only be exported/imported in read-only mode. That is, even if a data resource is exported it can still be edited in the xymphony from which it has been exported, and when a data resource is imported it cannot be edited in the xymphony to which it has been imported. Thus, at a given point in time there will be one and only one xymphony in which a particular data resource can be edited.

There are several ways in which the simplistic scheme outlined above could be enhanced. Rather than a xymphony simply making resources available in read-only mode to other xymphonies by exporting them, one could export resources to specific xymphonies only. The xymphonies to which one would like to export could be named on an individual basis, or

according to some more generic specification (e.g. related to the hierarchical structure of xymphonies).

Returning to our example scenario, consider Anjali and Bansi working in Bangalore. Anjali decides that she wants to share some of her documents with her collaborators in Oslo and Boston, and so she marks those documents as candidates for export to all xymphonies that belong to the same xymphonic project (there are only three xymphonies in this project, viz. Boston, Oslo, and Bangalore). If the xymphony owners (or their deputies) in Boston and Oslo choose to import some or all of Anjali's exported documents, they immediately become available for browsing (but not editing) by Arne, Berit, Alice, and anyone else who work in those xymphonies.

Similarly, if Bansi needs access to some documents from Boston, he can see what documents have been exported from the Boston xymphony, who is editing each one of them, and their status. Assuming he has the authority to import documents to the Bangalore xymphony, he chooses the ones he needs, and can then browse them.

What if more fine-grained control is needed? What if e.g. Anjali would like to give Bob (a colleague of Alice in Boston) and Berit, but no one else, access to the documents in the Bangalore xymphony? Rather than exporting any documents, she could then invite Bob and Berit as read-only participants in the Bangalore xymphony. What if Anjali would to like share her documents with someone, but Bansi is not yet ready to share his documents with anyone outside of Bangalore? Then Anjali could create a sub-xymphony just for her own documents, and then export them or invite read-only participants to her sub-xymphony.

More examples could be given. The point being made in this section is as follows: The xymphonic model offers great power and flexibility in this area. Implementations could range from simplistic to very sophisticated. Keep in mind, therefore, that even a sophisticated implementation in a complex environment will be easy to use. This is so because no human being *has to* keep track of all the dependencies that are established between xymphonies, resources, and people. It suffices to focus only on those xymphonies with which one is personally involved. On the other hand, people who *need* control and total overview will find a xymphonic system an indispensable tool (see Section 6.9 for an elaboration of this).

6.6 Division of work among collaborators

The point of collaborating in the first place is to have multiple people contribute towards a common goal. Thus, a collaborative system needs mechanisms for dividing up work among the participants. It has been illustrated above that xymphonies help with just that, and that concurrency control in a xymphony makes sure its resources are divided among the participants in the sense that their access to those resources is properly coordinated. However, these two aspects of a xymphonic system will take you only so far. One quickly realizes that a natural style of collaboration will require more flexible mechanisms for division of work. Let's have a look at some examples.

Consider once more Anjali and Bansi working in the Bangalore xymphony. They have divided the documents in that xymphony among themselves. Anjali is editing some documents, Bansi some others, and perhaps there are other people in Bangalore working concurrently with them in that xymphony. The concurrency control in the Bangalore xymphony eliminates the possibility of two or more people accidentally working on the same subdocument simultaneously.

This is fine as far as it goes, but there is a need for more *dynamic* mechanisms for division of work among the collaborators. What if e.g. Bansi needs to make some modifications to a document currently edited by Anjali? Should he ask her to terminate her transaction so that he

can get access? Such a brute force method would of course work, but it would not be a very elegant solution. It would for instance force Anjali to give up transactional control over a document the contents of which she is responsible for, giving her no control over what kinds of changes Bansi makes to it. The xymphonic model offers a much better solution, by allowing Anjali right there and then to create a sub-xymphony for Bansi's work. While Bansi is working in Anjali's sub-xymphony, she can continue working on any other document she is responsible for. When Bansi is done, he will have to commit his work to Anjali's sub-xymphony, allowing her to review his changes before accepting them. This process also gives Anjali a guarantee that the document will indeed be passed back to her in due time.

The power of the xymphonic model becomes even clearer when you consider collaboration among people working in different xymphonies in the first place. Let us assume that Berit in Oslo discovers that she needs a contribution to one of her documents in a particular topic where Anjali in Bangalore is an expert. One possible solution could be to invite Anjali as a participant in the Oslo xymphony, but that would potentially give Anjali access to more documents than is perhaps desirable. Also, this solution would force Berit to give up transactional control over her document, depriving her of the possibility of reviewing any and all changes before accepting them into her documents. It is therefore much better for Berit to create a sub-xymphony for Anjali. This sub-xymphony can then be copied to the server in Bangalore and execute there, or Anjali can work remotely within Berit's sub-xymphony in Oslo. In either case, Berit gets to review Anjali's contribution before accepting it.

6.7 Recursive division of work

We have seen above that the xymphonic model provides elegant support for sharing of information, geographical distribution of work, division of work, and delegation of work. In the last example Berit created a sub-xymphony that enabled Anjali to make a contribution to Berit's document. In the general case, any xymphony can have any number of sub-xymphonies, each one of which can again have any number of sub-xymphonies, and so on. This is an example of *recursion*, a powerful concept exploited in many areas of computer science.

The practical implications for people working in a collaborative project are attractive indeed. Not only can a project leader create a suitable number of xymphonies distributed over a network of servers, and not only will xymphonies help in ensuring that the right people get the right kind of access privileges to the right resources at the right times, xymphonies also support repeated and dynamic delegation of work through an arbitrary number of levels. The latter is made possible by recursion. Let's have a look at an example.

Consider Alice, Bob, and other team members working in the Boston xymphony. Let's say Bob has been assigned the responsibility for the contents of three chapters having to do with legal issues. After working on those chapters for a while, Bob has a much clearer picture of what their contents will have to be and what kind of legal expertise will be required to complete the work. He therefore decides to delegate responsibility for selected portions of the work to Peter, Paul, and Mary. After some time, Paul realizes a subsection, for which he does not have the necessary expertise, must be added to one of his documents. After a few phone calls, he finds that Tim is both capable and willing to make the desired contribution. Paul then creates a sub-xymphony just for Tim, and delegates to it the subsection in question.

Interestingly enough, Bob could be totally unaware of this delegation from Paul to Tim. Bob just knows that Paul is responsible for a certain portion of the work, and that the system will ensure that Paul eventually commits his work to Bob. Alternatively, if Paul's performance is inadequate, the system will allow Bob to abort Paul's work. Similarly, when Paul delegates to

Tim, Paul need not concern himself with whether Tim actually does the work himself, or delegates (part of it) to someone else.

Some readers may be concerned that in certain situations one might want to have tight control over how much delegation goes on in the system. The xymphonic model can easily allow many different control mechanisms to be introduced. For example, one could specify that a given xymphony cannot have sub-xymphonies at all. Alternatively, one could specify that only certain people be allowed to create sub-xymphonies. Moreover, one could specify a limit to the number of levels of sub-xymphonies that a given xymphony can have, or that the system as a whole can have. There are other possibilities as well. Thus, there is no reason why a xymphonic system should not have adequate control mechanisms in place, ensuring proper operations.

6.8 Persistent savepoints

The glossary of (Gray & Reuter 1993, page 1035) defines savepoint thus: "A named point in the execution of a transaction. In case of failure, the transaction may roll back to one of these savepoints and reestablish the transaction state as of that point. Savepoints may be *persistent.*" The meaning of the term persistent in this context is that the contents of the savepoint is permanently stored, and will not be lost even if the system is shut down or crashed.

Persistent savepoints is a prerequisite in any system that supports transactions that last for more than a few minutes. Thus, support for persistent savepoints is included in the xymphonic model. This ensures that minimal amounts of work will be lost in case of failure, and it allows users to go back to the state of the system as of some earlier point in time. The latter is somewhat similar to *versions*, which can be used to track the history of a data resource (see discussion in Section 7).

6.9 A dynamic information repository

It follows from the above that xymphonies form hierarchies (also known as tree structures), and that these can be arbitrarily large both in terms of breadth and depth. In such a hierarchy, a given person can be the creator of, as well as participant in, any number of xymphonies. The resulting web of interactions and relationships between xymphonies, people, and data resources can be quite overwhelming, for a human being, that is. Not because a xymphonic system is unnecessarily complicated, but because the way people prefer to perform collaborative work will in some cases be inherently complex.

The more complicated a collaborative project becomes, the more important it is that the project leader and participants have easy access to information. A xymphonic lock manager component constantly keeps track of the kind of information discussed here, and can at any time provide answers to a multitude of questions, including e.g. the following:

- Who is currently active in the Bangalore xymphony?
- Who is currently browsing (but not editing) documents in the Oslo xymphony?
- What documents are currently available for browsing or editing in the Boston xymphony?
- Who is the creator/owner of the Bangalore xymphony?
- Which xymphonies does Anjali currently own?

- In which xymphonies is Anjali currently active?
- In which xymphony is a given document currently editable?
- Who is currently editing a given document?
- What is currently the status, quality, reliability, or degree of completeness of a given document?
- What percentage of the documents in the Oslo xymphony is still labeled as incomplete drafts?
- For each document in the Oslo xymphony not yet approved by the project manager, retrieve the name of the person currently editing that document as well as its status.

The list could go on. Having answers to such questions available at one's fingertips is certainly attractive in any non-trivial collaborative project.

7 Comparison with currently used techniques

Today's collaborative products are typically based on one or more of the following basic techniques:

- **Locking**. In a lock-based system, if one person is working with a data resource, that resource is marked by the system as *locked* as long as that person is working with the resource. Locking systems can have different degrees of sophistication, but most will at least distinguish between read and write locks. Most transactional systems use some form of locks, but the use of locks does not necessarily cause a system to be transactional.

  The xymphonic model is based on locking. The flexibility and power of this model comes from its novel use of *recursive* locking as well as *parameterized* locks (see above).

- **Versioning**. A versioning system will allow a data resource to exist in multiple *versions*. If the system allows a sequence of versions (1, 2, 3, …) for a resource, these are known as *revisions*. If the system allows branching to take place, i.e., the system allows multiple versions to be derived from one particular version, then such multiple versions are known as *variants*. Variants may have to be *merged* (i.e., combined) later, to form a new version containing the changes from the different variants. Versioning is attractive for some application domains, e.g. where it is important to keep track of the *history* of resources.

  However, versioning is a poor mechanism for supporting concurrent collaborative work. This is so because of the problems related to merging the variants that result from such work: To the best of my knowledge, no *general* algorithm for merging exists (i.e., the merging process can only be automated in special cases), so human involvement in the merge process is often required.

- **Check-in/check-out**. In a check-in/check-out system, a data resource must be *checked out* before a person can start working on it, and the resource must be *checked in* when the work is completed. While a resource is checked out it is marked as such, i.e., it is essentially locked by a check-out lock. However, checking out a resource does not only involve locking it, but also creating a separate copy of it (it is then this

copy of the resource that can be worked on). During check-in, the original version of the resource is replaced by the modified version.

Carefully note that no more than one person can check out a given resource at a time. Also note that if someone tries to *read* a checked out resource, he or she will get access to the original version of the resource, not the separate copy currently being worked on by the person who performed the check-out operation.

Check-in/check-out is actually a combination of very simple locking and very simple versioning. This simplicity is probably its main virtue, and it is successfully used in many real-world applications. However, as this paper attempts to demonstrate, check-in/check-out is not at all as powerful as the xymphonic model.

The above techniques can be complemented by less basic techniques, such as e.g. a good user interface that makes it as easy as possible for users to exploit the available functionality. This is one of the reasons there are many proprietary solutions out there which work quite well for certain application domains. For example, Microsoft Word allows annotations and/or change proposals to be inserted into a document, thus creating a new version of that document. Such a version can then be merged with the original version, allowing the person in charge to accept or reject the various change proposals. Because of the nice user interface, this works well as long as no more than two people are involved. As the number of people involved grows, this style of collaboration gets more and more unwieldy; it simply does not scale well.

The following subsections in turn discuss the three basic techniques mentioned above.

7.1 Locking versus xymphonic collaboration

A xymphonic system is indeed a lock-based system, just much more sophisticated than most other such systems. The latter is obvious from the preceding sections of this paper, which describe functionality not found in other lock-based systems. And, of course, anything you can do in a purely lock-based system, you can also do in a xymphonic system.

7.2 Versioning versus xymphonic collaboration

Pure versioning is not at all a powerful collaborative tool. It does not help much when it comes to sharing information among team members, and it lacks the kind of control structures described in the preceding sections. For example, bare bones versioning has very little to offer in terms of distribution of work, division of work, and delegation of work. We suspect, therefore, that in practice vendors of versioning-based systems tend to add something extra, rather than providing just bare bones versioning. Some of the challenges involved in using versioning as the basis for collaboration are described in (Magnusson & Asklund 1995).

If you wanted to make a versioning-based system that was as powerful as a xymphonic system, you would have to add mechanisms for dealing with meta-information, and you would need a recursive control structure that regulates access to data resources, coordinates participants, and ensures proper commit and abort routines. In effect, you would have to introduce xymphonies, thus making your versioning-based system xymphonic. Probably a very good idea!

The one great thing that a versioning-based system has which a xymphonic system only support to a certain degree through persistent savepoints is versions (i.e., revisions and variants – see above). Revisions make up an historic track record for the work that has been done, and variants allow branching into multiple alternative versions. There is no denying that this is attractive for some application domains. However, as shown by (Bakken 2002), you

can have the best of both worlds by combining versioning and xymphonies. This is so because there is nothing in the Xymphonic Model that prevents having one or more of the resources in a xymphony being subjected to a versioning regime.

7.3 Check-in/check-out versus xymphonic collaboration

A xymphonic system supersedes a check-in/check-out-based system. First of all, whenever a participant in a xymphony works on a data resource, it is in effect checked out. This is so because the participant has a local copy on his or her machine, while the primary copy is locked on the server (but still available to browsers). Secondly, a xymphonic system supports what could be referred to as *recursive* check-in/check-out through its hierarchic structure of xymphonies (see above). In other words, delegating a resource from one xymphony to another can be regarded as a form of check-out.

In order to see why check-in/check-out is completely inferior to xymphonic collaboration, compare the capabilities of one with the other:

- **Sharing of information between readers and writers.** Check-in/check-out is primitive in this area, giving readers access to new information only after check-in is completed. The xymphonic model is much more flexible and customizable in this area, offering check-in/check-out semantics if that is desirable, and going far beyond that whenever needed.

- **Meta information.** The check-in/check-out model has nothing to offer in this area.

- **Dividing up work.** The check-in/check-out model supports division of work in the sense that once a resource is checked out by one person, another person cannot check the same resource out until the first person completes his or her work. Xymphonies provide a lot more power in this area.

- **Distribution.** There is no support for distribution in the check-in/check-out model per se. Xymphonies, on the other hand, are very natural units of distribution, facilitating load balancing, local work groups, and better fault tolerance.

- **Dynamic delegation of work.** The possibility of dynamically delegating work from one person to another whenever the need arises, is central to the xymphonic model, and is easily accomplished by creating new xymphonies. The check-in/check-out model has nothing to offer in this area.

- **Information about the work process.** A check-in/check-out-based system will let you know whether or not a given resource is checked out, and, if so, by whom. That's about it. The xymphonic model is much richer in this area, giving project leaders and system administrators the kind of tools they have been dreaming about but never thought would become reality.

Wait a minute! This is unfair, you might object at this point. There is nothing to prevent a vendor from extending a check-in/check-out-based system with better mechanisms in these areas. That is correct, but nothing short of xymphonic extensions would suffice. And that is precisely one of the points we want to make, that vendors of check-in/check-out-based software should start thinking of how they can evolve their products into xymphonic systems.

Unlike versioning-based systems, check-in/check-out-based systems have absolutely nothing to offer beyond what xymphonic systems do.

8 A few words about Xymphonic Systems AS

Xymphonic Systems AS is a small technology start-up established in 1998 (originally under the name of Apotram AS) by Telenor Research and Development. Telenor (www.telenor.com) is the largest telco in Norway. We currently have two US patents related to xymphonic transactions, both granted.

8.1 We have already implemented much of the above

Much of the functionality described above has been implemented by Xymphonic Systems AS, and has been available for demo purposes since 2002. In other words, this is not just "theory".

9 Conclusions

This white paper is about the xymphonic model of collaboration; its power, its elegance, and its potential to change the way we use computers. The lack of proper support for collaboration in today's computing infrastructure is undoubtedly one of its most profound shortcomings. Even with the best possible user interfaces to exploit the underlying functionality, combinations of locking, versioning, and check-in/check-out leaves a lot to be desired in terms of collaborative power. As a matter of fact, we have yet to encounter a knowledgeable researcher or IT professional that will not freely admit that there is large room for improvement in this area. This paper has shown that xymphonic collaboration is superior to the techniques currently used by commercial products.

10 References

Alonso, G, Agrawal, D, El Abbadi, A, Kamath, M, Günthör, R, Mohan, C. 1995. *Advanced transaction models in workflow contexts*. IBM Almaden Research Center, San Jose, California, IBM research report RJ9970.

Anfindsen, O J. 1995. Dynamic cooperation between database transactions by means of generalized isolation levels. In: *Proceedings of 2nd International Conference on Concurrent Engineering: Research and Applications*. McLean, Virginia, 249-260. (this paper was published shortly before the formal foundation of xymphonic transactions had been developed, but is included here in the interest of completeness)

Anfindsen, O J. 1996a. Cooperative work support in engineering environments by means of nested databases. In: *Proceedings of 3rd International Conference on Concurrent Engineering & Electronic Design Automation*, Poole, UK, 325-330.

Anfindsen, O J. 1996b. Cooperative work support in multimedia conferences by means of nested databases. In: *Information Network and Data Communication*. (Proceedings of 6th IFIP/ICCC INDC International Conference, Trondheim, Norway) F A Aagesen, H Botnevik, D Khakhar (eds). London, Chapman & Hall, 317-326.

Anfindsen, O J. 1996c. Supporting Cooperative Work in Multimedia Conferences by means of Nested Databases. In: *Proceedings of Norwegian Informatics Conference - NIK'96*, Alta, Norway, 311-322.

Anfindsen, O J. 1997. *Apotram - an application-oriented transaction model*. PhD Thesis, Department of Informatics, University of Oslo, Norway. Research Report 215 (this paper defines xymphonic transactions, the academic name of which is Apotram). Available in electronic form from the author.

Anfindsen, O J. 1998a. Conditional Conflict Serializability - an application-oriented correctness criterion. In: *Proceedings of International Workshop on Issues and Applications of Database Technology - IADT'98*, Berlin, Germany. 47 - 54.

Anfindsen, O J. 1998b. Conditional Conflict Serializability - an application-oriented correctness criterion (extended version). *Journal of Database Management,* Vol 9, No 4, 22 - 30.

Anfindsen, O J. 2002. *Collaborative work and xymphonic transactions.* White paper. Available in electronic form from the author.

Bakken, S. 2002. *The xymphonic transaction model and versioning – how can the xymphonic transaction model benefit from incorporating versioning concepts?*. Master thesis, Department of Informatics, University of Oslo, Norway.

Gray, J, Reuter, A. 1993. *Transaction processing: concepts and techniques*.SanMateo, Calif., Morgan Kaufmann Publishers.

Kaiser, G E. 1990. A flexible transaction model for software engineering. In: *Proceedings of the International Conference on Data Engineering*, 560-567.

Kaiser, G E. 1995. Cooperative transactions for multiuser environments. In: *Modern database systems*, Kim, W (ed). Reading, Mass, Addison-Wesley, 409-433.

Korth, H F, Speegle, G. 1990. Long-duration transactions in software design projects. In: *Proceedings of the International Conference on Data Engineering*, 568-574.

Korth, H F, Speegle, G. 1994. Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model. *ACM Transactions on Database Systems*, 19, (3), 492-535.

Kumar, M, Wong, J. 1993. Transaction management in design databases. *Journal of Systems Software*, 22, 3-15.

Magnusson, B, Asklund, U. 1995. Collaborative Editing - distribution and replication of shared versioned objects. In: *Proceedings of ECOOP'95 Workshop on Mobility and Replication*. Aarhus, Denmark.

Nodine, M, Zdonik, S. 1992. Cooperative transaction hierarchies: Transaction support for design applications. *VLDB Journal*, 1, (1), 41-80.

Rauft, M A, Rehm, S, Dittrich, K R. 1990. How to share work on shared objects in design databases. In: *Proceedings of the International Conference on Data Engineering*, 575-583.