OVERVIEW

1 Introduction

Synchronous cooperative applications (synchronous groupware) allow a group of users to work jointly and simultaneously on shared data (such as a document). In contrast to, for example, a flight booking system, the actions of each user will instantly be visible and effective for every other user of the application. For applications under consideration here, a high degree of synchronicity is important i.e. the actions of one user should be immediately perceived by other cooperating partners. It is not sufficient simply to observe changes to the common data source. Information about the context of the change is also needed regarding who initiated the action, and if possible, why it was initiated. Conversely, the users should be aware that their own actions can have effects on other users of the system. This perception of what other members of the group are doing is known as group awareness, and serves the purpose of helping users to coordinate their actions within the group. The COAST framework intends to make the development of synchronous groupware easier, with a complicity comparable to the development of equivalent single-user applications. This is achieved by supporting the groupware developer on the technical as well as on the conceptual side.

2 System architecture

The COAST architecture is made up of two components: clients and mediators.

We have chosen a replicated architecture in which each user interacts with one application instance, called a client. This approach allows keeping application semantics to the client. Furthermore, this helps in load balancing. Shared objects from the shared object space are replicated on demand for each client to meet the responsiveness criterion. COAST additionally allows the state of replicas at individual clients to diverge over periods of time. This enables the application to give optimal responsiveness to write operations taking the risk of late rejects.

Mediators are responsible for the persistent storage of the shared object space and for the synchronisation of all existing replicas. They maintain the primary copy of the data that is shared among their clients. Application data needed when running a COAST application is transparently loaded/replicated on demand from a COAST mediator on a cluster by cluster basis (a cluster contains a bunch of objects and is the smallest unit of replication). One role of the COAST mediator is similar to that of a WEB/FTP server namely to retrieve data from a persistent storage and send it over the network to a client upon request. The COAST mediator additionally provides services for creating new clusters or volumes in its namespace (volumes consist of a set of clusters). The COAST mediator is a generic component and as such independent from specific

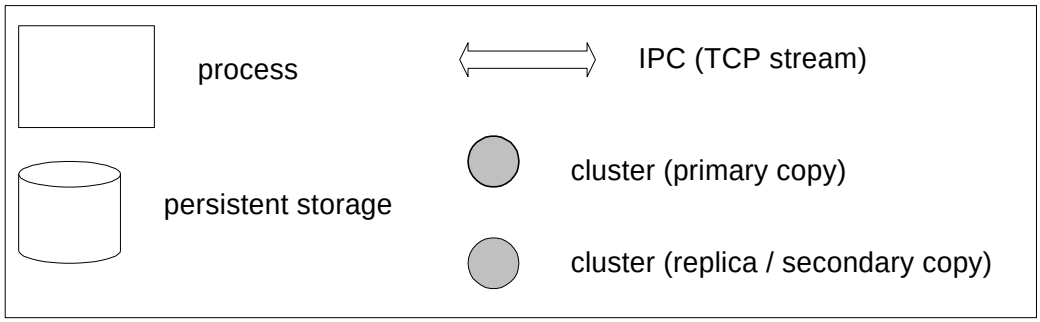
application data. Note: A COAST Client can also obtain data from the local file system. In this case no mediator is needed. However, sharing of local data among different clients is not possible.

Network connections from clients to mediators are established on demand whenever clients want to retrieve a part of the shared object space available at a mediator. Connections are kept open for synchronisation purposes until the part is not needed any more.

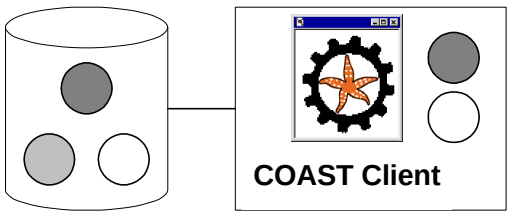
A distributed transaction mechanism (i.e. clients and mediator are involved in transaction processing and concurrency control) providing conventional and optimistic transactions keeps the shared object space in a consistent state and cares for persistency.

2.1 Architecture examples

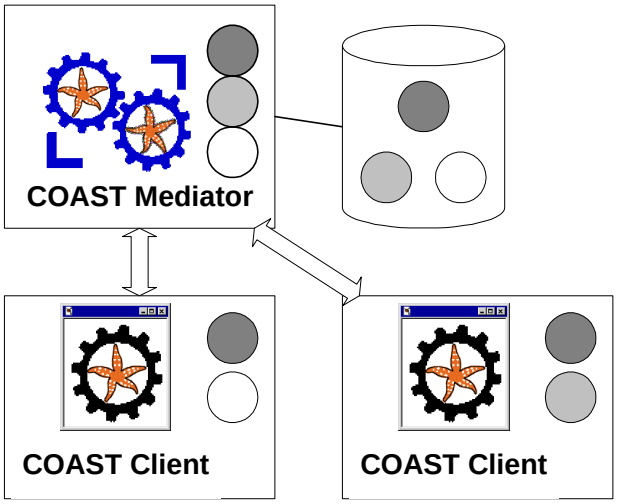
In the following, three example architectures are shown. Big boxes in the pictures denote potentially different sites/processes in a networked computer environment. Arrows denote communication channels between processes. Small circles represent clusters where circles with the same shading are different replicas of the same cluster.



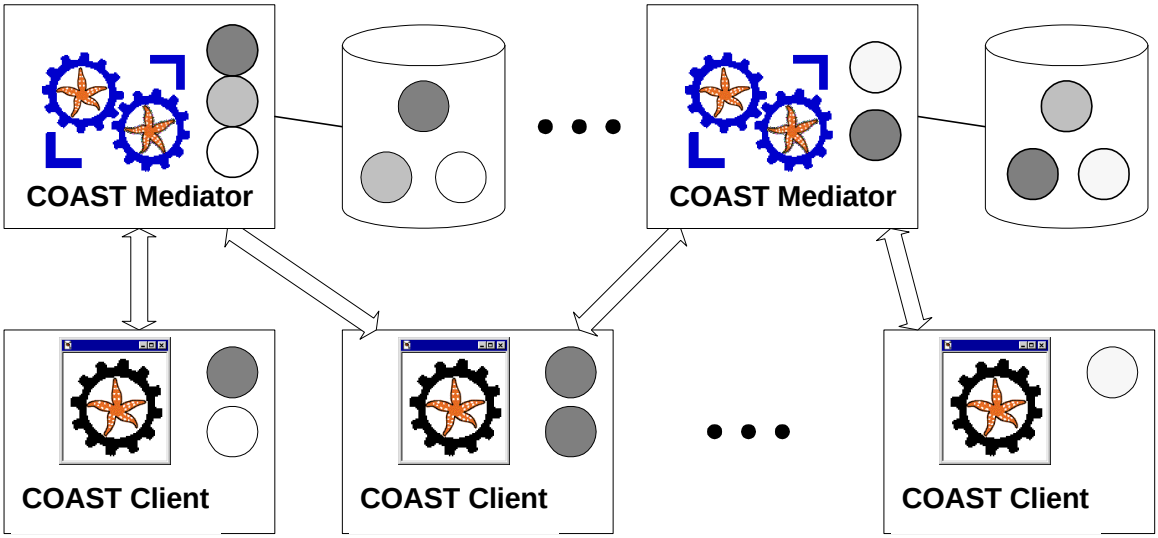
notation



single-user version - suited for one user, no shared object space



multi-user version - suited for medium-sized object spaces and a few users



multi-user version – suited for large-sized object spaces and many users

2.2 Robustness

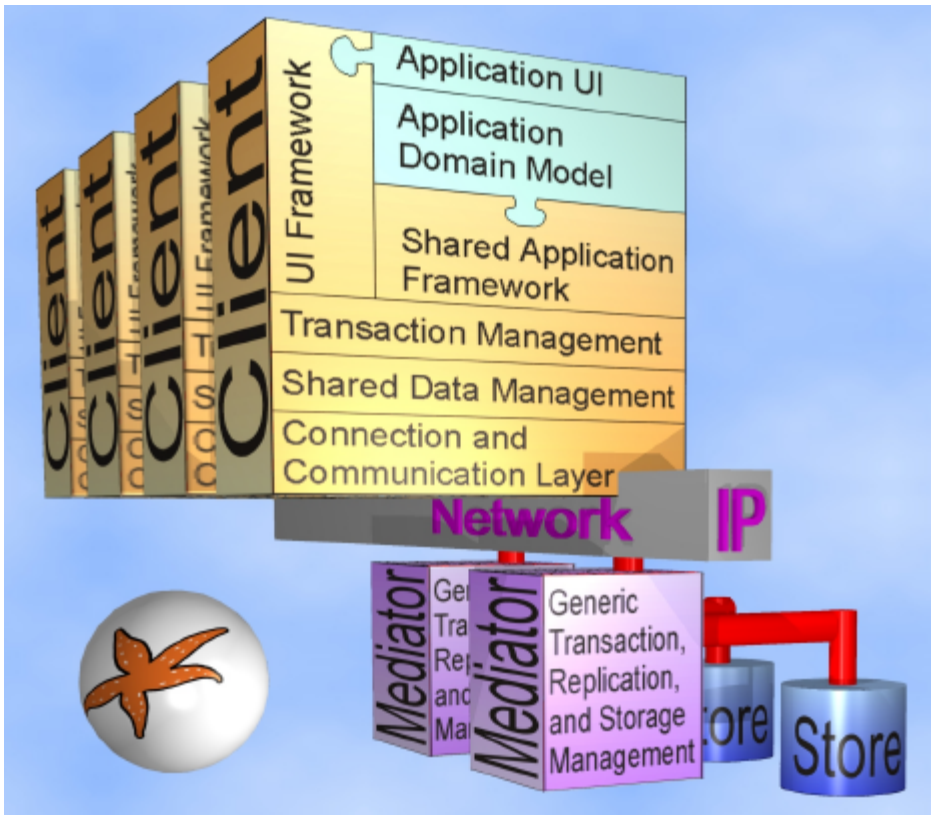
Regarding robustness, the crash of a client has no effect on any other component. No data is lost as the client holds only object replicas.

A crashing mediator results in the sudden unavailability of the corresponding parts of the shared object space. Clients currently using these parts would run out of synchronisation. Therefore every visualisation that makes use of these parts is automatically closed. Clients may retry to access these parts as soon as the crashed mediator is up again or use another mediator after the corresponding volumes have been relocated.

Depending on the mediator's persistency strategy (e.g. a mediator can be configured to store after every nth transaction) more or less data will be lost.

2.3 The fancy picture

The following fancy picture illustrates the system architecture and the layered software architecture together. You can see the fat clients (full application logic at client site) with the generic COAST parts in yellow and the application-specific parts in green. Note that mediators do not contain application-specific parts.



3 Shared objects

The COAST-system provides a shared object space where arbitrary, application-specific model objects can be shared among locally distributed users. Depending on the application shared object spaces can range from single documents (e.g. a piece of text, a calendar, etc.) to very complex information spaces (e.g. a hypermedia document, a virtual world, etc.).

Programmers can define, instantiate, access, and modify shared objects as if they would be virtually local objects. That means, the COAST framework cares about replication, synchronisation, concurrency control, and persistent storage of shared objects.

3.1 Frames

COAST uses frames to model application objects of the shared object space. Frames were originally used within the domain of artificial intelligence for knowledge representation. Relationships between frames are not just like references in ordinary programming languages. Supplementary they can carry semantics such as inversion and conclusion rules.

One main goal of a frame system is to add more semantics to object-oriented object structures in a declarative manner. Normally there is very little semantics the programmer can add to an object's attributes. In typed programming languages the type (or class) of an attribute can provide some clue about object structures. In untyped programming languages like Smalltalk there is no semantics at all despite from the attribute's name that may help persons understand an attribute's purpose. A frame system allows the application programmer to declare properties and constraints such as inversion and cardinality constraints for object attributes. These 'enhanced' attributes are called slots and objects using slots are called frames. A frame system consists of everything needed to declare and use frames.

The frame system monitors access to frames and slots to provide the declared properties or check constraints. That means the frame system has full control over the state of all frames. Based on this fact, the frame system can provide additional generic facilities such as transactions or persistent storage that can be used for any user-defined frame. The COAST frame system uses inheritance to gain control on all slot accesses.

3.2 Replication and synchronization

For reasons of data availability and modification responsiveness parts of the object space are replicated by COAST. Using an optimistic concurrency control COAST allows the state of replicated data to diverge over periods of time. COAST's synchronization mechanisms however, always ensure a non-diverged and consistent object space as soon as possible (regarding network delays and processing load). Replication and synchronization of application data is completely handled by COAST.

The smallest unit of replication is the cluster, a collection of model objects.

3.3 Structuring the object space

Two mechanisms allow application developers to structure the shared object space:

- Shared objects can be grouped to clusters. Clusters are the smallest unit of replication and should include objects that are usually used together. Clients working on different clusters do not interfere with each other.

- Clusters can be grouped to volumes. Volumes provide a logical abstraction from actual mediator addresses. A COAST mediator can serve one or more volumes at a time. For load balancing or administrative purposes it is possible to flexibly assign volumes to mediators and thereby configure the system in an optimal way.

As a shared object space grows in size, more mediators can be added and volumes can be re-assigned.

The structure of the shared object space's content (inside structure) has implications on its cluster/volume structure (outside structure). However, inside and outside structure are handled independently. For the generic architecture components only the outside structure is important. The individual application mainly deals with the inside structure. As already mentioned, shared objects are loaded on demand. This mechanism makes object loading as well as cluster loading transparent. As far as the process of cluster loading may include the establishment of new client-

mediator communication channels (as shown in figure 7), these are transparent to the application developer, too. There are only two cases in which application programmers deal with outside structure:

1. For newly created objects, the application developer may specify a target cluster and volume. The application developer may also create new clusters and volumes. The framework can not handle this structuring task because optimal outside structure depends on the application semantics and thereby the inside structure of the shared object space.

2. For objects that constitute an entry point into the object net, a naming service is implemented that allows retrieving named objects from within a volume. This service is essential to provide initial access to a shared object space.

4 Transaction management

Transactions play an important role in COAST. They allow application programmers to define consistent changes of the state of the shared object space.

Traditional transaction management would be too slow for interactive applications that make use of explicit application dynamics modelling. In such applications every action changing the user interface state (e.g. cursor moving, scrolling, pointing, etc.) results in a change of the shared object space. For this reason, COAST introduces optimistic transactions. Optimistic transactions are based on a client's replicated objects only, assuming later mediator commit. The optimistic transaction is then reported to the mediator which may decide, sometimes with significant delay, if the transaction is to be committed or rejected. In the worst case, a whole stream of optimistic transactions may be rejected resulting in an undo of the corresponding user's actions.

Of course, such behaviour is not always adequate. Often the user wants to be sure that his or her action will not be undone a few seconds later. Therefore COAST also offers conventional (i.e. pessimistic and slow) transactions for critical operations. Given a specific application context it is up to the application programmer to decide between responsiveness and safety, i.e. a transaction has to be fast (optimistic) or safe (pessimistic).

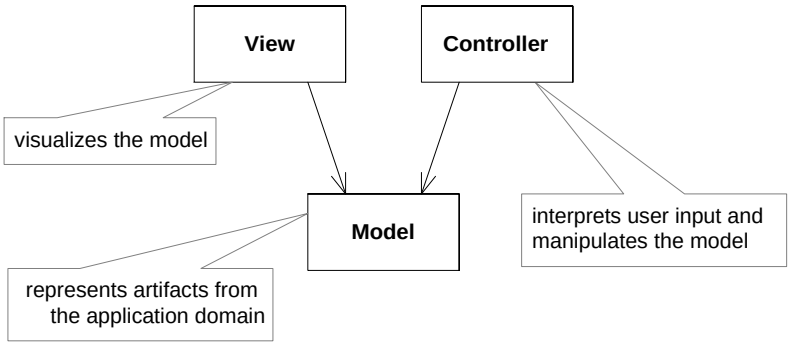
Consistency of a shared object space has to be guaranteed independent from its structure. As a result the COAST framework also allows transactions in which more than one mediator is involved. These transactions are necessary when objects originating from several mediators are modified in the course of a transaction (e.g. when establishing cross-volume object references).

Note, that transactions in COAST are intended to be short because the effect of a transaction is not visible to the outside until transaction commit. Long transactions would result in poor group awareness as changes to the shared object space would be invisible to other users while the transaction is running. Moreover, a complex state change resulting from one long transaction is more difficult to understand by other users than the equivalent subdivision into several shorter transactions.

5 Modelling groupware

To maintain a consistent and flexible visualization of shared objects, concepts like *Model-View-*

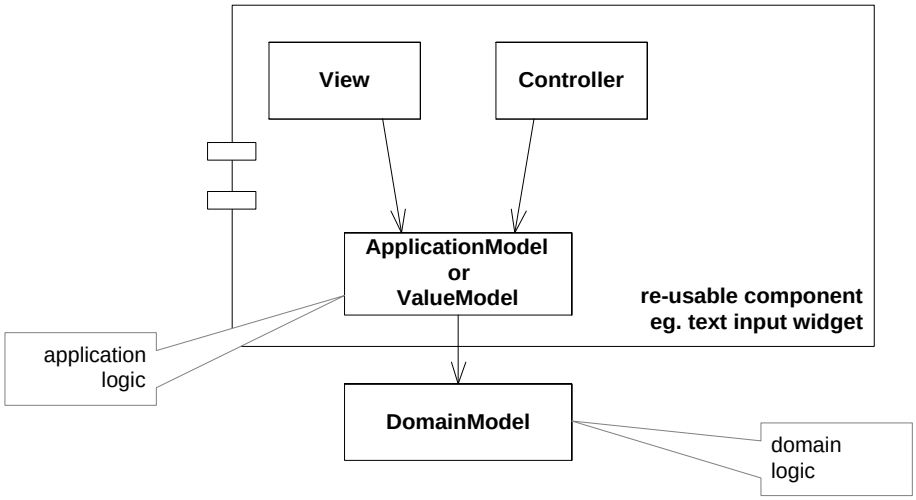
Controller have already been extended for the use in groupware applications. These approaches usually operate on the micro level of modelling like MVC does. This means, they propose how shared model objects can be attached to multiple views and controllers at different sites, but do not include an overall application model that describes how the application as a whole is composed.



Classical Model View Controller

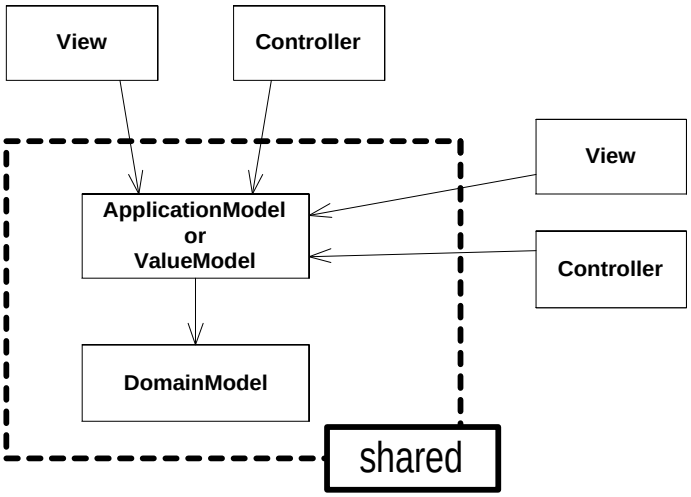
However, with shared objected spaces and associated visualization concepts solving fundamental issues, it is possible to define a *uniform* application model for multi-user applications on a more abstract level. Such a model has to consider groupware-specific aspects like group awareness or floor control as well as object-oriented software engineering-specific aspects like usability and reusability.

In the design of single-user applications the separation of *application model* and *domain model* has proven successful.



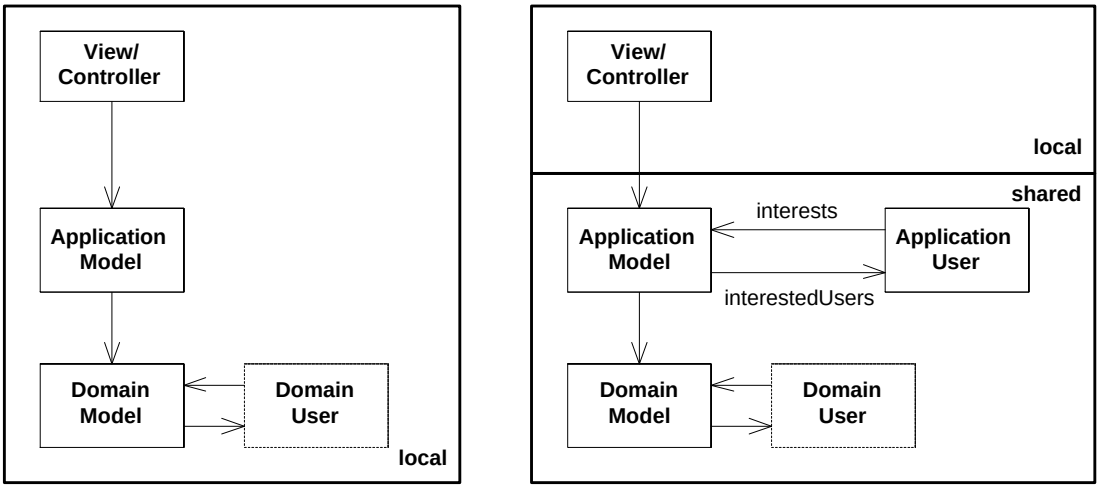
Better re-usability by introducing an application or value model.

In our approach, we try to transfer this concept to collaborative applications. We claim that this separation is helpful for the development of multi-user application as well, if it is based on a shared object space and adapted to the specific needs of the multi-user case.



Domain and application models are shared objects

Blah blah.



Comparison between conventional model for single user applications (left side) and our model for groupware applications (right side).

DETAILS AND ADVANCED FEATURES

1 Persistent storage

Shared objects in COAST are automatically stored persistently in the file system. Volumes are represented as directories of the same name and clusters appear as files in the directory of their corresponding volume.

The storage interval (autosave feature) of a volume can be set to ensure storage after N modifications to a cluster. Minimum for N is 1, which would result in immediate storage after every transaction. The default value for N is 10000, which appears to be quite high but can be reached fast for highly interactive applications with many short transactions in short time periods. N can be set by sending the message `autoSaveInterval:` to the cluster manager. Note, that this message has to be used on the mediator's site to have effect there.

When the last client to a specific volume disconnects (or the volume is not used any more in the single-user case) it is always completely saved.

A special configuration file ('volume.ini') is used to store meta information about the volume. Two parameters in this file can be modified by editing the file:

- `marshallingType=binary / ini` Sets the cluster storage format. Binary is the default, ini is a readable ascii representation for debugging purposes.
- `useGarbageCollection=true / false` Turns garbage collection for the volume on (default) or off.

2 Memory management

The COAST framework is equipped with several mechanisms for memory management regarding the mediator or client's main memory and disk space usage. Furthermore, caching facilites reduce the network load for replication.

2.1 Garbage collection

Everytime the last used cluster of a volume is released, the cluster manager responsible for this volume (this is the mediator or the local cluster manager) will start a Coast Garbage Collection (CGC) for this volume. Everything reachable from any named model in this volume will be considered as alive frames, the rest will be removed. For technical reasons, no cross-volume

references will take affect to the result of the CGC so the Coast application must ensure the availablity of those references.

In some cases, it might be useful to disable the CGC (e.g. if it is assured that there will be no garbage at all). To do so, you can add the line "`useGarbageCollection=false`" to the corresponding volume.ini-file.

2.2 Cluster unloading

During a coast session, many clusters can be loaded. To reduce the load of the client, some of them are unloaded. First, it's important to know that some of the clusters can not be unloaded since they hold models needed to compose the views. From the remaining "unloadable" clusters some can be unloaded. This is done by default with a Least Recently Used - Strategy (based on last timestamp a cluster was used). It's possible to change the strategy or to configure a given strategy. The additional choices are Not Frquently Used (based on the times a cluster is used), Extended Working Set (based on the distance to some unloadable Frames) and Second Chance (LRU + trying to avoid multiple loads/unloads) and the configuration options are the amount of clusters, when to try an cluster unload (default: 30) and the amount of clusters to keep (default: 20)

2.3 Caching

Everytime a cluster is requested (implicit by accessing yet unreplicated model of it), It has to be transmitted from the mediator to the client. Thus it is desirable to save the cluster local in the cache-directory and check on requesting, if the cached cluster can be reused. When a cluster is released (by shuting down the application or by an unload strategy) it will be copied to the cache (as long as caching for clusters is enabeld). If you can foresee that there will be only few cachehits, you can disable caching.

For Binary Large Objects (BLOBs, used with the CoastBinaryContainer) caching is even more efficent since there are much more cachehits and much higher performance gains. To take this fact in account, you can also limit the caching to BLOBs only.

3 Constraints in COAST

3.1 Local objects coexisting with shared objects

Besides shared objects, there are local objects which represent local displays and which are responsible for the interpretation of local input events. These objects also interact with local system objects representing the local infrastructure such as the window system or audio devices. If any information represented in the shared object space is shown on a local display, it is some local user interface objects' responsibility to transform this rather abstract information into a visual form and to accept graphical interaction related to this display.

Local objects are bound to a site's local environment (i.e. output and input devices) and thus can not be part of the shared object space. However, the need for local objects leads to interactions between local objects and shared objects. From an information flow point of view these interactions can be seen in two directions:

1. Interpreted user input flows from local objects to shared objects. In order to realize this flow of information, it appears necessary that local objects can make calls to shared objects. This raises no problems.

2. Information that has to be displayed flows from shared objects to local objects. This also applies whenever information held in shared objects changes. Therefore, it appears to be necessary that shared objects can send messages to local objects. This appears to be a problem, because local objects do not have a globally unique identity and shared objects cannot send messages to objects that they do not know.

To solve this problem (among others) the COAST framework uses a one-way constraint system. In a one-way constraint system the application programmer is declaring dependencies rather than programming update methods. We call these dependencies *application programmer defined constraints*. One advantage of a constraint system is that the flow of information from shared objects to local objects can be realized without sending messages. Once there are local objects being dependent on shared objects it is the task of a system contained constraint engine to realize the flow of information. The constraint system will organize this flow of information after local objects have been created. The question remaining is how to create these local objects as a result of change in the shared object structure (e.g. in order to open a window on a remote computer). To answer this question, we introduced the concept of system immanent constraints, which are always present and which we distinguish from application programmer defined constraints.

3.2 System-immanent constraints

We define a user's presence as the relation between a user and the site (i.e. workstation) at which the user is currently working. Using the information about the presence of a user-immanent constraint, we define the following system:

1. For each presence of a user, media outlet channels (e.g. windows or audio channels) have to be open for each interest relation between the shared object representing the user (the user model) and a shared object representing the application (the application model).

2. Every open channel has to show or play up-to-date information as referenced and as defined by application dynamics at any time.

These constraints should be maintained by a constraint engine which opens and closes local media outlet channels and thereby creates, removes, and updates local objects in response to the change of shared objects.

3.2.1 The constraint engine

To fulfil the constraints listed in section 'System-immanent constraints' a constraint engine has been installed on the client side. This constraint engine consists of two parts:

- A so-called interest monitor is responsible for starting and terminating applications

according to the user's interest in shared application models.

- A notification mechanism notifies observers of shared objects whenever the latter change. In addition, the COAST framework automatically tracks and maintains these observers.

Remember that the application programmer must follow a constraint-based programming paradigm. Application programmers never explicitly handle visualisation processes; these are always triggered by changes of the shared model in which case programmer-defined visualisation rules are used.

3.3 User Notification

Given a constraint system that follows the system immanent constraints as described above, it is possible to notify other users without inter-process message passing or event mechanisms.

A piece of information (e.g. a reminder or message) has to be combined with an application for displaying it (e.g. a sound player or message window) which is then simply added to a user object's interests. The notification will reach the user as soon as possible (i.e. as soon as he or she is connected to the shared object space).[1]

4 MVC

GUI-objects under COAST can often be implemented by using the modified user interface builder. If the application programmer wishes to implement his own GUI-objects, he has to know the following.

The COAST framework is based on the *MVC-paradigm* (MVC = Model View Controller) which means that GUI objects are considered as view objects and controller objects. View objects are responsible for displaying a model (document) object on the screen. Controller objects are associated with view objects and are responsible for interpreting interactive user input attached to the view. As a result of an interpretation, controller objects typically send messages to model objects in order to cause a change of the document structure.

In COAST, there are two major rules, which the application programmer has to follow when programming views and controllers:

- View objects must be implemented according to a special updating scheme, which is based on declaring functional dependencies.
- When controller objects interact with model objects (documents) in order to cause change, the messages sent have to be encapsulated in transaction parenthesis.

---

[1] Etwas unklar

The COAST cooperative MVC-paradigm differs from the conventional MVC-paradigm in many aspects.

When implementing view objects under COAST,

- the application programmer does not implement any kind of code responsible for updates (such as the `update:` methods),
- the application programmer does not explicitly register the view to be dependent on other objects (e.g. on model objects),
- the application programmer does not explicitly invoke any kind of updating mechanism (such as the so-called `changed:` methods).

Instead, the application programmer divides his view variables (slots) into two categories:

- slots containing redundant values, which can be computed from the other values (e.g. contained in the model),
- slots containing basic values, which store non-redundant information and which are not dependent on other values and therefore do not have to be updated as a reaction on change of other slots.

Slots of the first kind are to be declared by the application programmer together with a method computing the value of this slot. The COAST system will ensure that any time this value is read it will contain the valid value according to the computation method.

This method can either be a formalized method allowing the system to efficiently perform incremental updates of the value or an arbitrary Smalltalk method. In both cases, the system automatically tracks all dependencies to other slots and is thereby able to perform updates rather efficiently. In other words, the COAST system allows the application programmer to assume that the functional relation declared by the computation method will always be true.

A special feature of the COAST view system is that also the display area of a view itself is considered as a redundant value, computed by the method "`displayOn: ...`". Thereby, the display itself is being updated the same way as dependent slots are, without any effort for the application programmer.

This results in a new philosophy for display programming. No effects on the display are explicitly caused. Generally, data is displayed because there IS a display consisting of windows, views and subviews and because these components depend on the shared objects, which may have changed. Each view displays data on the screen not because the application programmer calls display functions but simply because it exists. Each subview exists because its container (the view or window which contains the subview) has requested its existence. The initial cause of this proliferating chain of cause and effect (i.e. the existence of the various components) is the window itself.

Consequently, in COAST windows are not opened explicitly by the application programmer. Opening a window explicitly would be an imperative style of causing effects on the display. The

fundamental idea is the presupposition that, like all other effects on the display, a window is there because it is implied by the application model.