Supporting the Construction of Distributed, Interoperative, User Interface Applications

Krishna Bharat

A Ph.D. Proposal

ABSTRACT

Distributed applications are hard to construct. They are both hard to design and hard to implement. Part of the complexity is due to the lack of the right programming abstractions -- remote procedure calls and sockets are low level and too far removed from the needs of user interface developers to be useful. It is also in part, due to the lack of suitable application design tools in the distributed arena. Specifically, tools which integrate the various tasks involved in developing user interface applications, and also support the right structuring abstractions needed to build distributed applications.

The goal of this thesis is to:

- Motivate the need for tools and abstractions that bridge the gap between the application programmer's needs and the facilities provided at the operating system/network level,
- Provide paradigms for structuring distributed, interoperative UI applications, and
- Show how these may be couched in an integrated application development environment, with abstractions to make programming and deploying a distributed application easy.

CONTENTS

1. Introduction

Although the operating system and networking support for distributed computing has been around for several years now, we have yet to see a proliferation of multi-user, distributed applications. This is because building distributed applications is hard. The investment in time and effort required seems justifiable only for large systems. Whenever possible, people tend to use pre-existing mechanisms such as *talk*, *nv*, *vat* and *e-mail,* instead of building their own custom tools.

Often distributed applications do more than just help people communicate; they allow people to collaborate on a shared context, and hence can require arbitrary kinds of interaction and dependencies between application components. Hitherto, programmers have relied on message passing or remote procedure call style mechanisms to program such systems. These mechanisms are low level and too far removed from the needs of distributed application developers to be effective. It has been the practice (in the C community in particular), to implement new facilities in the form of libraries, rather than as language features. Indeed there are good arguments for keeping system related issues outside the language. However, this usually burdens programmers with the task of mapping their mental model to lower-level operations. The higher the level of abstraction provided, and the better it is integrated with the language, the easier this task becomes.

Direct manipulation GUI builders have proven very successful in supporting and integrating the interface design, programming and testing phases of application development. However, the program structuring paradigms they employ do not scale to a distributed setting. To be truly effective, the application builders themselves need to be aware of the distributed nature of the applications being built. For instance, a “distribution-aware” builder, which understands the difference between *local* and *global* resources, would be able to structure the programmer’s code and data-structures in a manner that best matches their usage pattern. It could leverage upon the distributed features of the language and library suite, and generate code for communication, synchronization and replication.

There is clearly a need for tools, abstractions and programming paradigms to bridge the gap between what application developers need and what programming environments are able to provide. In this proposal we examine a few approaches that seem to show promise.

2. Approach

We have a two-pronged approach to supporting the construction of distributed applications.

First, based on our experience with application designers for single-user applications -- notably Visual Basic -- we designed and implemented an **application builder cum execution environment** for distributed applications called Visual Obliq. Applications may be interactively designed, programmed and tested within a single tool. As in Visual Basic, the builder generates stand-alone code in an interpreted language -- in our case, Obliq[12], a distributed scripting language. The builder is distribution-aware in that it provides support for structuring code, spreading the session and migrating computation over the network. The application developer is given a simple programming model to work with; a model which hides the distributed nature of the application by providing transparent access to remote objects, and a simple API for explicit network-related operations, whenever necessary.

Secondly, we propose a set of **paradigms** for programming distributed applications. These are more high-level than rpcs and message-passing approaches that are currently in vogue, and are sufficiently abstracted from system dependencies that they may be couched in a programming language framework. Our paradigms fall into two categories: (a) Scope based approaches, and (b) Constraint based approaches. In addition we describe the implementation of two lower-level abstractions, (i) Active Fields, and (ii) State Pickling, which may be used to implement some of our paradigms as well as others.

3. Development Environment Support

The Visual Obliq[3] application builder (Fig. 1) supports the ‘Shared Scope’ paradigm for the construction of distributed applications (described in the next section). The builder allows the user interface to be drawn and programmed by the application developer. It comes with a simple API and distribution support for spreading the session from site to site. All parts of the application are globally accessible from all sites. This makes programming a multi-user, application no more difficult than programming a conventional, non-distributed application.

Figure 1: The Visual Obliq application builder

3.1 End User's Perspective

The *end-user* is the person who runs a distributed application designed in Visual Obliq. Starting an application called `foo` on a machine called `bar`, creates a multi-user session with the public name `foo@bar`. Other participants could either *join* the session on their own accord, by typing "`join(foo@bar)`" within a Visual Obliq interpreter, or be *invited* to the session by other members. Invitations are extended with the help of a *Visual Obliq Agent* running on the invitee's machine, which pops up a notice on the user's machine to find out if they wish to be connected to the session. Once the user is connected to the session, the code and state needed to join the session comes over the wire. Visual Obliq applications can be arbitrary multi-user, user interface applications. The functionality at each site need not be the same, allowing for the creation of *heterogeneous* applications.

3.2 Application Developer's Perspective

The *application-developer* is the person who builds the Visual Obliq application using the VO application builder.

First, in the interface design phase, a direct-manipulation interface builder is used to draw the user interface for the windows in the application. Each such window is called a *form*. Forms nest the widgets available in the builder's palette. Various attributes may be set via an accompanying

Forms are converted by the builder into programmable objects in the language, Obliq. The names given to widgets in the design phase are mirrored in the Obliq object. In the programming phase, the programmer can extend each form-object with additional fields and methods, including callbacks for each widget it contains. Each form can be multiply instantiated at run-time creating multiple instances of the form, possibly in different address-spaces.

In addition to form-specific code, the programmer can add *global code*, that is visible at all sites. The global code contains by default, an array for each form designed by the programmer. Each such *form-array* contains instances of the various instances of the form at run-time. Since form-arrays are globally visible, any form-instance can be accessed from any part of the system via its form-array.

The builder generates a procedure called a *session-constructor*, used to spread the session to new sites. The session-constructor can be customized by the programmer to decide which forms are initialized at each site. This procedure is transmitted to the site of a user who either *join*s a session or accepts an *invitation* to a session. To invite the user at site `bar` to the session, the programmer uses the command: `installAt(bar)`.

A second class of applications can be created in Visual Obliq: *Migratory Applications*, which are arbitrary single-user applications that can move to a new host upon execution of the command, `migrateTo(hostname)`. This command can be invoked at any point within a callback. The semantics of executing the `migrateTo` command are as follows: *If the operation succeeds,* the entire application will move to the new machine and exit the callback. The application will terminate in the original machine. *If the operation fails* (due to a network exception), the application will continue its execution locally, starting with the instruction following the `migrateTo` command.

4. Paradigms

Next, we list a few programming paradigms that we have had the opportunity to implement and study. This list is by no means exhaustive. No single paradigm will suffice in all cases, and sometimes a combination of approaches may be required.

4.1 Scope Based Approaches

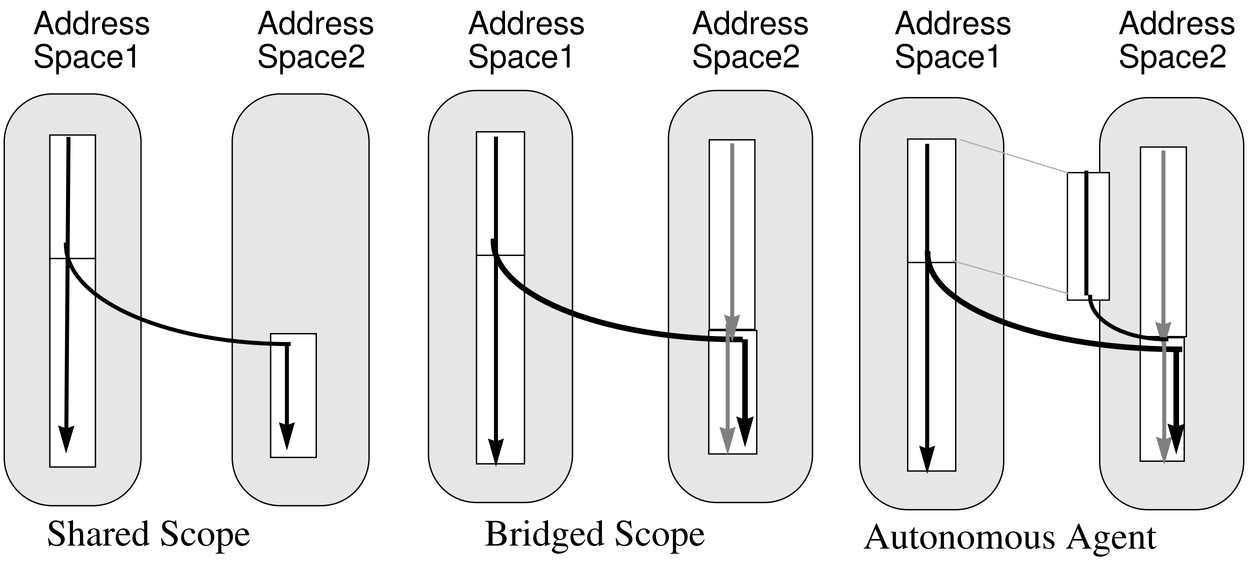

Figure 2: Various Paradigms based on Scope

4.1.1 The 'Shared Scope' Paradigm

In this programming model (supported in Visual Obliq[3]), the application starts at one site and spreads to other sites by transmitting some of its code and part of its scope. All sites share a portion of the application's initial scope, which contains global code and data, and implements a global name-space. The unit of distribution is a form (a programmable window object). Forms in other portions of the distributed application can be accessed via the global name-space, and operated upon transparently. The session spreads by the transmission of an abstraction called the 'session constructor' which brings over the necessary code to a new site, as well as the shared scope. The constructor then instantiates forms.

This approach was found to be well suited for launching and sustaining distributed sessions on the web (described in [4]).

4.1.2 The 'Bridged Scope' Paradigm

In the previous case, the application was required to evolve from a single site. In the 'Bridged Scope' case, the application transmits part of its computation and scope to a peer, to be executed within the confluence of the two scopes.

This approach was taken in Visual Obliq (described in [3]) for the implementation of *remote callbacks*. A callback designated as "remote" by the programmer is packaged in a procedure, to be shipped to the destination host when invoked. At the destination, an object specified by the programmer then executes the callback-procedure, passing in a handle to the local environment as the variable `REMOTE`. This allows the callback to utilize resources in both scopes.

4.1.3 The 'Autonomous Agent' Paradigm

This is a variant of the 'Bridged Scope' approach. Here the transmitted computation brings over all the state and code that it needs, allowing it to break ties with the original site. This is done by bringing over a set of handles to the relevant application state and 'deep copying' all the needed data-structures and code over. Note that we generalize the notion of state to include code as well. In a statically scoped language a piece of code could have references to other pieces of code and data, which constitute its scope. Similarly, data can have references to code. In any application, we can conservatively define the *relevant application state* for a given thread (namely the state relevant to future execution of the thread) as the portion of the application state visible to the thread. In more practical terms this is the state that is accessible from the thread-stack by following a sequence of references. In a single threaded user interface application, the only executing thread is the user interface thread. When not engaged in executing the callback, the stack of the user interface thread only has references to the windows in the system, and transitively, the state that hangs off them. This vastly simplifies the task of identifying the state to be copied when migrating a single threaded UI application.

In a user interface application, the migrating agent will need to copy over windows as well, with their state intact. This poses an interesting practical problem because much of the interface state is kept externally, in UI-libraries or in the window server. Thus the application needs a way to checkpoint its user interface state into application objects, destroy the windows, transmit its application state and rebuild the UI from the saved context.

The amount of state that is transmitted will decide how much of the application will move to the new site. In the limiting case, when all active windows migrate, the agent may be the entire application! This was the case in our implementation of 'Migratory Applications' in Visual Obliq[5].

4.2 Constraint Based Approaches

These are radically different from the scope-based approaches described so far. A constraint, in the most general case, is a relation between data-values. A constraint system enforces a set of constraints over a set of data-values. Shown below are a pair of constraints that relate the bottom-right corner of a rectangle to its top-left corner and its width and height.

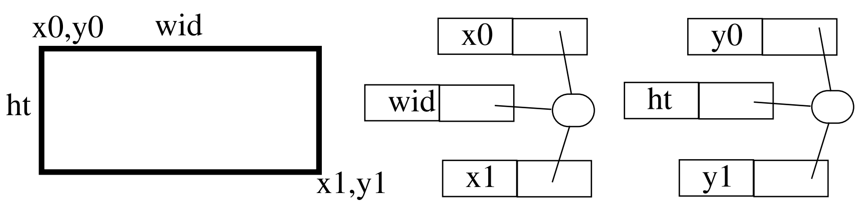

Figure 3: Constraints

4.2.1 Distributed Concurrent Directed Constraints

In this paradigm, application state resides statically in individual address-spaces, but data-values are connected by a graph of directed, inter-address-space constraints. A *directed constraint* is one in which the parameters to the constraint are divided into two sets - inputs and outputs, and the constraint sets the value of the outputs based on its inputs. Output variables cannot be changed by other agencies, such as other constraints, but input variables can be.

Figure 4 shows a directed constraint graph in which the width and height have been made dependent upon the corners of the rectangle. The width and height cannot be changed directly. Instead when the corners get new values (e.g. during interactive resizing), the width and height will be automatically updated.

Although not as expressive as general-purpose constraint graphs, these are often sufficient for user interface applications and quite efficient to implement.

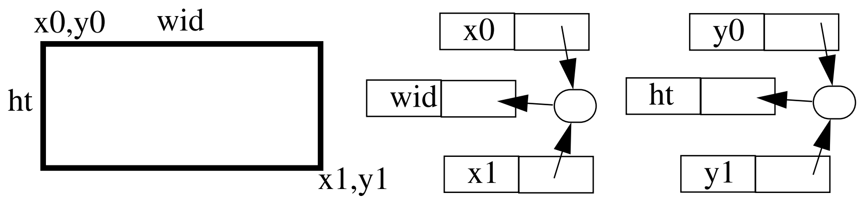

Figure 4: Directed Constraints

A **distributed constraint graph** would allow a set of applications to execute cooperatively by maintaining a sequence of relationships between values situated in distributed contexts. For instance, this would allow changes in data in one or more address-spaces to be reflected in graphical views in other address-spaces, thereby supporting visualizations of distributed state. While the use of constraints to maintain relationships *within* a user interface application has been well researched, the use of constraints to maintain links *across* applications is relatively new.

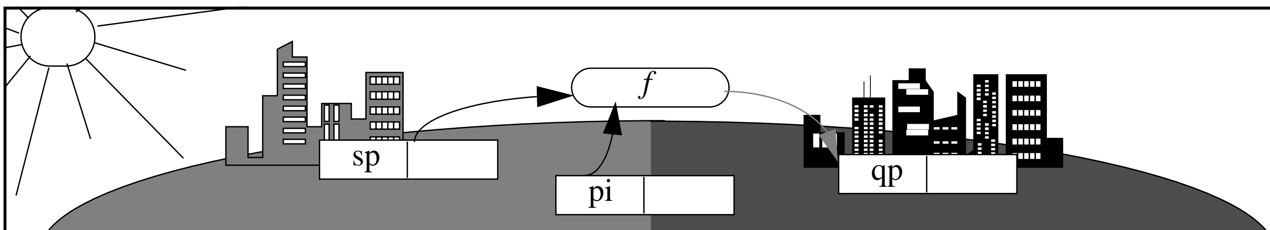

Figure 5: Distributed Constraints

The hard problem (it turns out) is not merely building such constraint systems, but implementing them efficiently. This is because concurrent updates often tend to overlap and interfere, and the constraint graph is itself subject to change. Previous distributed approaches[17] have been synchronous, allowing no concurrency whatsoever. The distributed constraint graph is a major bottle-neck, and traditional constraint maintenance algorithms scale very poorly. To address this, we have developed a highly efficient, distributed algorithm for concurrent, one-way constraints, called Doppler[6]. The algorithm requires no global synchronization or locking. It tracks causal relationships between graph updates, and allows updates to proceed concurrently whenever possible. In terms of concurrency, the algorithm is provably optimal.

4.2.2 Intuition Behind Doppler

We describe briefly the intuition behind the Doppler algorithm:

There are two approaches one can take for value propagation in a directed constraint graph:

(a) **Eager**, which is data-driven and causes all dependent variables to get (potentially) new values when a variable is written to, by recursively evaluating downstream constraints*, and*

(b) **Lazy**, wherein dependent variables are merely invalidated upon a write (no constraint functions are evaluated), but a read to an invalidated dependent variable causes its associated constraint to be evaluated on it inputs. For this purpose, the input variables are read recursively.

In the eager method, there is one catch however --- the constraints need to be evaluated in topological order. In a distributed setting, since the graph may be changing all the time, keeping track of the topological order is a hard problem. Hence we preferred the lazy approach.

In implementing a concurrent version of the lazy algorithm, we can take the following steps to improve performance:

(i)We can execute writes in the background, in a non-blocking fashion. Hence a thread doesn't have to block for writes.

(ii) We can allow concurrent writes to execute together because although they may overlap, but since they perform identical (hence commutative) operations on variables, they do not interfere.

(iii) We can allow concurrent reads to execute together for the same reason. For efficiency, we might want the threads to lock the nodes they read, so that the work is not duplicated unnecessarily.

Reads and writes cannot execute together because the read may base its value on a partially invalidated graph, and hence reflect neither the state of the graph after the write, nor the state before. *Hence if a read needs to reflect a certain write, it needs to block for the write to finish*.

In a distributed system, given a pair of truly concurrent operations, their ordering in real-time is indeterminate within the system, and usually irrelevant. This allows us to order a concurrent reads and a write in an arbitrary fashion or not at all (i.e. we can let them remain concurrent). In fact we can avoid blocking the read by allowing it to return a value based on the state of the graph before the write occurred. Of course, this may require increasing the book-keeping in every node to allow the read and write to co-exist and see different views of the graph. This is a space vs. time trade-off to increase concurrency and hence performance.

Thus the semantics of a read is as follows: *It returns a value based on finished writes and ignores all concurrently executing writes*.

However, a read cannot always choose to ignore a concurrent write. There are some cases in which a read must reflect a concurrently executing write, and hence needs to wait for it to finish. These cases all have to do with **causality**

Causality stated simply says,

- A read must reflect every write that could potentially have caused it. In general, since we do not have specific knowledge of the computation being performed, we must assume that any write *known* to the thread at the time of the read could have caused it.
- If a read reflects a certain write, *w*, it must also reflect every write that could potentially have caused *w*. Again this means it must reflect every write that was *known* to the writing thread when it executed *w*.

When we speak of knowledge in the context of threads, we use the term in its broadest possible sense. If a thread perceives some consequence of a write, albeit in an indirect manner, it is said to have knowledge of the write. The best we can do is get an over-estimate of the knowledge that threads have of writes in the system. To do this, we conservatively track information flow in the system. A thread obviously knows about every write it has initiated. When a thread communicates with another, we shall assume that it could have passed on information about all the writes it knows about.

Figure 6 shows two examples of *read-write* causality. In case i (the degenerate case), the thread initiates the write W_1 and then executes the read R_1. The thread would naturally expect R_1 to reflect W_1. Hence the read is made to block.

In case ii, the read is executed after T_1 sends a message (or executes a remote procedure call) to T_2, denoted by the gray arrow. This communication constitutes a causal link, since it could have conveyed information about W_1 that prompted T_2 to execute R_1. Hence, R_1 must wait for W_1 to finish, and then compute its value.

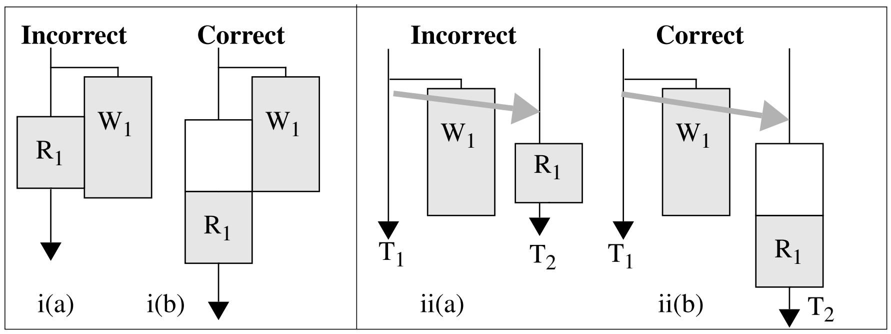

Figure 6: Reads that must follow causally preceding writes

Figure 7 shows an example of *write-write* causality. As before the gray arrow represents a message between threads. The write W_2 may causally depend on W_1 because a message travelled from T_1 to T_2 in the interval between the writes. In a more general case there could be a sequence of messages from T_1 to T_2, through a number of other threads, which could causally influence the execution of W_2.

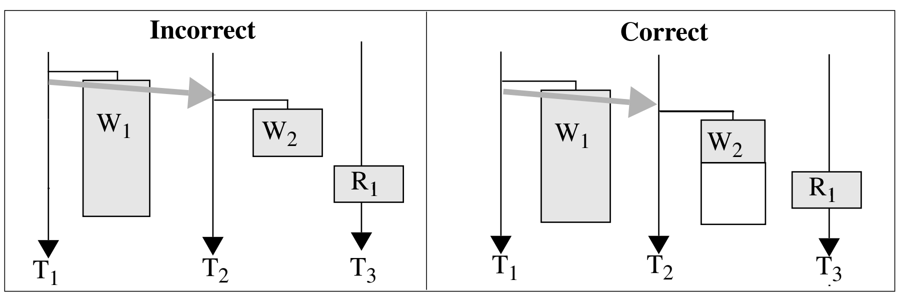

Figure 7: Writes that need to follow causally preceding writes

This causal link does not prevent W_2 from executing concurrently with W_1 but it does prevent it from being seen before W_1. If it completes early (as in the left half), then the read R_1 could potentially reflect W_2 without reflecting W_1. This would cause the thread T_3 to see the effect without seeing the cause, which would be inconsistent. Hence in a correct execution the commit point of W_2 will be delayed until W_1 finishes and R_1 will reflect neither. A degenerate but very common case of causality is when both writes are executed by the same thread. The second write could be based on the assumption that the first had occurred.

Our distributed algorithm is implemented by each of the thread constraint systems. Each constraint system tracks two monotonically growing sets of writes - the set of *causally preceding writes* and the set of *writes that are known to have finished*. The first set can be tracked accurately, by attaching causality estimates to user-level writes. The set of finished writes is maintained in constraint systems, by periodically exchanging their estimate of finished writes with peers.

A pre-requisite for the correct execution of reads is that the read must wait for all causally preceding writes to finish before beginning. This happens when the writes that were in the causally preceding set *at the time the read was initiated*, become contained in the finished writes set. Likewise a write can finish only after causally preceding writes have finished.

The semantics of a read is as follows: It will be ordered after (and return a value based upon) all writes that are known to be finished or causally preceding at the time of initiation.

The semantics of a write is similarly as follows: It will be ordered after (and be seen to happen after) all writes that are known to be finished or causally preceding at the time of initiation.

Note that these conditions impose a partial ordering on writes and reads. We call this the *Global Partial Order*.

The book-keeping in the nodes of the graph is no longer a single value or invalidation mark; instead it is a set of values and invalidation marks partially ordered by the ordering of the writes which caused them in the Global Partial Order. This is needed to ensure that reads and writes execute correctly and retain the semantics described above.

In [6] we describe how the sets are maintained efficiently, and how reads and writes are actually executed. Also, we explain how the book-keeping data can be garbage collected over time and how graph-mutations are handled concurrently.

In terms of concurrency this algorithm provides *optimal concurrency for any lazy algorithm for directed constraints*. Writes execute in the background and do not block the executing thread. A read may block for causally preceding writes. If it did not then it would either return an invalid value or violate causality. Hence this blocking cannot be avoided in any scheme. Under all other circumstances reads and writes execute concurrently, unaffected by other operations, as in the sequential scheme.

4.2.3 Comparison to Distributed Shared Memory Systems

A **causal constraint graph** (CCG) is quite different from a **causal distributed shared memory system** (CDSM), although both of them are shared data-structures that provide causal read-write semantics.

Our implementation of causal constraints tracks causality due to both internal and external information flow. Internally, once a read reflects a certain write, the write will causally precede all future operation. External communication, as in user level messages, may convey information about writes known to the sender-thread. This is tracked as well. In CDSM systems, typically only internal information flow is tracked.

The semantics of sharing and replication is different in each case. In CCGs, each variable belongs to a single thread, which alone can read from it and possibly write to it (if it is an input variable). In CDSM systems all variables are globally readable and writable. However, in CCGs there can be arbitrary relationships between variables that cause a write to one variable to result in updates to a (continuously changing) set of other variables.

The two abstractions are different enough for us to treat them as distinct mechanisms for sharing state. Although, in principle we could use one to build the other, the implementation would be highly inefficient.

4.2.4 Concurrent Undirected Constraints

Our eventual goal is to layer an undirected constraint algorithm, such as DeltaBlue[13], on top of Doppler. There are two parts to implementing such a constraint algorithm. An *incremental constraint planning phase*, which orients, removes or enforces constraints when the structure of the graph changes. This in essence, converts an undirected constraint graph into a corresponding directed constraint graph. The second part consists of doing *value propagation* in the directed graph, for which Doppler will suffice. We have yet to devise an algorithm for doing the incremental constraint planning in a concurrent fashion. The fact that Doppler can give us global-information about the graph (hitherto unavailable) may possibly help us in this venture.

5. Lower Level Abstractions

Now we describe in brief, some lower-level abstractions that we studied to a limited extent. These may be used to implement some aspects of the paradigms described in the previous section and others as well.

5.1 Active Fields

An active field is a field in an object that can trigger a callback in response to a read or a write. Specifically, it can be configured to invoke a callback in response to one or more of the following events: `BeforeRead`, `AfterRead`, `BeforeWrite` and `AfterWrite`. The callbacks are executed in a blocking fashion. Fore instance, the `BeforeRead` callback is guaranteed to execute *after* the read is initiated but *before* it actually happens. Trapping the `BeforeRead` event allows a field to be updated lazily -- only when needed by a read. The `BeforeWrite` event is useful for synchronization, e.g. when a lock needs to be acquired before the write can happen.The `AfterWrite` event is useful for performing some invalidation or updates in response to a write or releasing a lock acquired before the write. `BeforeRead` and `AfterRead` can be used for synchronization as well.

5.1.1 Implementation

In a language which allows references to primitive types, it is often not possible to determine at compile time whether or not a given L-value is a certain field within a certain object --- and hence whether it is an active field or not. It is possible to do this at run-time by performing additional checks during assignments and reads; specifically by comparing the value with the memory addresses of known active fields. However, this approach is likely to be very expensive, since the check is necessary for all instances of the field's type (e.g. all integers). Instead, if we allocate objects containing active fields separately, on a protected page, then we can rely on the processor and operating system to notify us before any operation on the fields of the object. Thereupon, the protection on the page may be temporarily turned off, and the operation can be allowed to go through, with associated callbacks being invoked at appropriate points. While this increases the cost of access to objects with active fields, accesses to other objects incur no penalty. This was the approach we took in our implementation of active fields in C++. The approach is largely independent of the language and requires only the cooperation of the memory allocator. Hence it is portable to other programming environments as well.

5.1.2 Application to Constraints

Active fields may be used to integrate constraints into an object system in a transparent and convenient fashion. In previous implementations, assignments and reads to constrained fields required explicit method calls to notify the constraint system of the action. Instead if such a field were *active*, then the `AfterWrite` event could be made to automatically induce a wave of invalidation (or value propagation), and the `BeforeRead` event could be used to invisibly refresh the value in the field if necessary.

5.1.3 Implementing Shared Widgets

Active fields can be used to maintain replicated copies of the same object with a variety of consistency semantics, suitable for the implementation of shared user-interface components needed in distributed applications. We implemented shared widgets (such as shared sliders) by subclassing the widget from the `SharedObject` class and having each copy of the widget notify and update all peers when its value changed. The `SharedObject` class overrides the default `new` method and has its memory allocated on a protected page. Further, the `SharedObject` class maintains for each object, a list of its peers, and provides a mechanism for replicating fields which subclasses can use.

5.2 State Pickling

State pickling is an automated mechanism for writing out the state associated with a given data-structure to a buffer, in a manner that allows it to be converted to an equivalent data-structure at a later point in time. The ‘pickled’ state in the buffer may be conveniently stored on disk, replicated or transmitted over the network --- operations which are more difficult to perform on the original data-structure. Pickling, like garbage collection, requires the tracing of references within data-structures and hence requires the support of the language runtime. Hence pickling is typically implemented as a runtime extension, as in the case of Modula-3. We had the opportunity to add pickling support to the Java language runtime [34] at Sun Microsystems Labs (East), based on our experience with the Modula-3 implementation. The details of this implementation cannot be described at the moment due to patent applications which are in progress.

5.2.1 Issues

The following issues need to be addressed in the design of a pickling mechanism:

- **Customization** - The default mechanism, which traces all references and writes out all fields as they are, should be customizable by the programmer, using ‘specials’. The specials should not only depend on the type of data-value being pickled but also the type of pickling scheme being employed -- e.g. a checkpointing pickler may behave differently than a marshalling pickler since the latter might employ remote-references in some cases.
- **Type-Safety** - There should be type-checking support to insure that pickled objects are only unpickled in environments where identical types exist.
- **Flexibility** - The programmer should have the option of pickling a certain type as a supertype and unpickling it as subtype, to ‘port’ data-structures to new formats.
- **Safety** - The pickling mechanism should employ checks and guards to prevent untrustworthy specials from misbehaving by: (a) Accessing hidden state by pickling, (b) Forging objects by unpickling from a concocted pickle, (c) Reinitializing objects by re-unpickling them, and (d) Sabotaging the pickling mechanism by not allowing supertypes a change to pickle/unpickle their types and hence creating half-baked objects.

5.2.2 Application to Agent Programming

Agent programming in general, and the ‘autonomous agent paradigm’ (described in 4.1.3) in particular, can make good use of pickling support to transmit the agent’s state. Our implementation of migratory applications in Obliq made use of Obliq’s `copy` mechanism, which in turn employs Modula-3’s `Pickle` package.

One of the problems faced in migrating an agent’s state is identifying the boundary of the agent. Not all data is relocatable. Some items may need to replaced by local values at the destination. If a customizable pickling scheme is employed the programmer can write specials for fields that are not directly relocatable. In particular, these specials can help identify the boundary of the agent-state, allowing the agent to leave one environment and ‘dock’ into another, without losing its identity.

6. Related Work

There have been a number of systems that allow users to construct user interfaces by direct manipulation, such as Menulay[10], DialogEditor[11], Druid[33], Gilt[26], and the NeXT Interface Builder[28]. Some such as Visual Basic[25], provide support for adding code to widgets within the same framework. Visual Obliq is the only application builder that we are aware of, which does this for distributed applications.

There has been some work in the area of *groupware toolkits*, to provide abstractions needed to build distributed applications. In Rendezvous[30], the programmer ties together values in multiple address-spaces using a “sharing object”, which serves as a central copy. In GROUPKIT[32], programmers use message passing style communication and shared widgets to build multi-user applications. Fresco[23], a distributed UI toolkit, allows the distribution of user interface components using CORBA. GUI components have IDL interfaces so that they can be accessed remotely, allowing interface components in different address-spaces to be pieced together to form a hierarchy. In none of these cases is the toolkit couched in a distribution-aware application builder.

Application migration most closely resembles the work on process migration in Operating Systems [31, 39], although the aim behind process migration is to do load-balancing and improve parallelism. Process migration is usually implemented at a low level, making no assumptions about the application structure, or even about the programming language. The machine architecture and environment are assumed to be very similar. Process migration is unable to cope directly with applications that keep part of their state externally, e.g. when much of the user interface state information is stored within a window system server (as in the X window system). Our support for agent migration most closely resembles the API for transmitting agents in General Magic’s MagicCap, namely the semantics of the `go` command in Telescript[38]. Not enough is known about the workings of MagicCap to provide a detailed comparison. It is not clear whether their agent migration strategy will extend to arbitrary applications. The biggest win of our design is that *any* single-user application in Visual Obliq can be turned into a mobile, interactive agent by invoking the migration command.

Java[34] and Tcl[29] have support for code-shipping over the network. The biggest difference between the execution of external programs in these systems and the migratory applications in Visual Obliq is that in the former case the applications always start from a default (start) state when they arrive at an interpreter, instead of continuing from where they left off as in our case. Hence it would not be possible to forward an arbitrary program to a new user after interacting with it, as one could with a piece of annotated electronic mail.

Constraint systems have been heavily studied as user interface specification techniques, and have been used in a number of systems to support a range of different user interface implementation tasks (for example [2, 8, 9, 13, 14, 15, 16, 17, 18, 21, 27, 35]). See [37] for a full discussion of the benefits of constraints. Constraint systems are uniquely suited for supporting multiple views of shared data, since they allow the relationships between data and user interface presentation to be described declaratively. Based on these descriptions, a constraint maintenance system is typically able to update (multiple) individual views automatically in response to any and all changes in underlying data objects.

As user interface applications start to become network aware and support collaborative work, they will be increasingly be called upon to operate in a concurrent and distributed fashion. Recent UI toolkits have begun taking a multi-threaded approach (e.g. Trestle[24] and Fresco[36]). Unfortunately, the update algorithms used by existing constraint systems were designed to be sequential, and cannot typically operate in a multi-threaded (concurrent) or distributed setting. (An exception is [17], which is distributed, but does not support substantial concurrency.) As a result, presently, the substantial advantages of constraints cannot be applied to distributed applications.

The Doppler constraint propagation algorithm described in this paper is based on a slightly simplified version of the efficient sequential algorithm described in [19]. Like its predecessor, Doppler is both incremental and lazy. In addition, the algorithm is inherently multi-threaded, and is designed to allow a very high degree of concurrency. Doppler delivers causal consistency. Causal consistency is based on Lamport's concept of potential causality [22]. It has been widely studied in distributed systems (e.g. causal broadcasts [7] and causal memory [1]), and is accepted as a good combination of concurrency and correctness. It allows more concurrency than conventional sequential consistency, wherein all operations are ordered linearly (often in an ad hoc manner). At the same time cause and effect relationships are respected.

7. CONCLUSION

Currently, programming environments and paradigms for building distributed user interface applications are virtually non-existent. In my research, I have explored a few approaches to solving this problem, and described some that seem to work well. Almost all the work mentioned here has been implemented and published.

Here are some achievements in the course of this research:

- Visual Obliq is the first ever application builder for distributed applications.
- Visual Obliq includes the first ever implementation of application migration at the language level.
- Doppler is the first ever algorithm for concurrent constraints. It provides optimal concurrency for any lazy algorithm for directed constraints that respects causal dependencies.

I wish to gratefully acknowledge the significant contributions of Marc H. Brown, Luca Cardelli, Mustaque Ahamad, and my advisor, Scott E. Hudson.

My expected date of graduation is June 1996.

8. REFERENCES

[1] Mustaque Ahamad, Gil Neiger, James E. Burns, Phil H. Hutto, and Prince Kohli, “Causal Memory: Definitions, Implementations and Programming”, To appear in *ACM Transactions on Distributed Computing Systems*.

[2] L.A. Barford and Brad Vander Zanden, “Attribute Grammars in Constraint-Based Graphics Systems”, *Software - Practice and Experience*, 19:4, 1989, pp. 309-328.

[3] Krishna Bharat and Marc H. Brown, “Building Distributed, Multi-User Applications by Direct Manipulation”, *Proceeding of UIST ‘94*, Marina Del Rey, CA, Nov. 1994.

[4] Krishna Bharat and Luca Cardelli, “Distributed Applications in a Hypermedia Setting”, *Proceedings of International Workshop on Hypermedia Design ‘95*, Montpelier, France, Jun. 1995.

[5] Krishna Bharat. and Luca Cardelli, “Migratory Applications”, *Proceedings of UIST ‘95*.

[6] Krishna Bharat and Scott E. Hudson, “Supporting Distributed, Concurrent, One-Way Constraints in User Interface Applications”, *Proceedings of UIST ‘95*.

[7] K. Birman, A. Schiper, and P. Stephenson, “Lightweight Causal and Atomic Group Multicast”, *ACM Transactions on Computing Systems*, 9:3, 1991, pp. 272-314.

[8] Alan Borning, “The Programming Language Aspects of ThingLab: A Constraint-Oriented Simulation Laboratory, *ACM Transactions on Programming Languages and Systems*, 3:4, Oct. 1981, pp. 353-387.

[9] Alan Borning and R. Duisberg, “Constraint-Based Tools for Building User Interfaces”, *ACM Transactions on Graphics*, 5:4, Oct. 1986, pp. 345-374.

[10] W. Buxton, M.R. Lamb, D. Sherman, and K.C. Smith, “Towards a Comprehensive User Interface Management System”, *Proceedings SIGGRAPH ‘83*, July 1983, pp. 35-42.

[11] Luca Cardelli, “Building User Interfaces by Direct Manipulation, *Proceedings of UIST ‘88*, Banff, Alberta, Canada, Oct. 1988, pp. 152-166.

[12] Luca Cardelli, “Obliq: A Language with Distributed Scope”, *Computing Systems*, MIT Press. 1995, 8(1), pp. 27-59.

[13] B. N. Freeman-Benson, John Maloney, and Alan Borning, “An Incremental Constraint Solver”, *Communications of the ACM*, 33:1, Jan. 1990, pp. 54-63.

[14] B.N. Freeman-Benson, “Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming”, *Proceedings of OOPSLA/ECOOP ‘90*, 1990, pp. 77-88.

[15] Michael Gleicher, “A Graphics Toolkit Based on Differential Constraints”, *Proceedings of UIST ‘93*, Atlanta, GA, Nov. 1993, pp. 109-120.

[16] Tyson R. Henry and Scott E. Hudson, “Using Active Data in a UIMS”, *Proceedings of UIST ‘88*, Banff, Alberta, Canada, Oct. 1988, pp. 167-178.

[17] Ralph D. Hill, “The Rendezvous Constraint Maintenance System”, *Proceedings of UIST ‘93*, Atlanta, GA, Nov. 1993, pp. 225-234.

[18] Scott E. Hudson and Roger King, “Semantic Feedback in the Higgens UIMS”, *IEEE Transactions on Software Engineering*, 14:6, Aug. 1988, pp. 1188-1206.

[19] Scott E. Hudson, “Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update”, *ACM Transactions on Programming Languages and Systems*, 13:3, July 1991, pp. 315-341.

[20] Scott E. Hudson, “A System for Efficient and Flexible One-Way Constraint Evaluation in C++”, Georgia Institute of Technology -Graphics, Visualization, and Usability Center Technical Report GIT-GVU 93-15, April 1993.

[21] Scott E. Hudson, “User Interface Specification Using an Enhanced Spreadsheet Model”, *ACM Transactions on Graphics*, 13:3, July 1994, pp. 209–239.

[22] Leslie Lamport, “Time, Clocks and the Ordering of Events”, *Communications of the ACM*, 21;7, July 1978, pp. 558-565.

[23] Mark Linton and Chuck Price, “Building Distributed User Interface with Fresco”, *Proceedings of Seventh X Technical Conference*, Jan 1993, pp. 77-87.

[24] Mark S. Manasse, and Greg Nelson, “Trestle Reference Manual”, *Digital Systems Research Center - Technical Report 68*, Palo Alto, CA, Dec 1991.

[25] Microsoft Corporation, “Microsoft Visual Basic”, *Microsoft Press*.

[26] Brad A. Myers, “Separating Application Code from Toolkits: Eliminating the Spaghetti of Callbacks”, *Proceedings of UIST ‘91*, Hilton Head SC, Nov. 1991, pp. 211-220.

[27] Brad A. Myers, Dario A. Giuse, R.B. Dannenberg, Brad Vander Zanden, D.S. Kosbie, E. Pervin, A. Mickish and P. Marchal, “Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces”, *IEEE Computer*, 23:11, Nov. 1990, pp. 71-85.

[28] NeXT Inc., “NeXTStep and the NeXT Interface Builder”, 900 Chesapeake Dr., Redwood City, CA 94063.

[29] John K. Ousterhout, “Scripts and Agents: The New Software High Ground”, *Invited Talk at the Winter 1995 USENIX Conference*, New Orleans, LA, January 19, 1995.

[30] John F. Patterson, Ralph D. Hill, Steven L. Rohall, and W. Scott Meeks, “Rendezvous: An Architecture for Synchronous Multi-User Applications”, *Proceedings of CSCW ‘90*, Oct 1990, pp. 317-327.

[31] M. Powell and B. Miller, “Process Migration in DEMOS/MP”, *Proceedings of 9th ACM Symposium on Operating System Principles*, 1983, pp. 110-119.

[32] Mark Roseman and Saul Greenberg, “GROUPKIT --- A Groupware Toolkit for Building Realtime Conferencing Applications”, *Proceedings of CSCW ‘92*, Nov 92, pp. 43-50.

[33] Gurminder Singh, Chun Hong Kok, and Teng Ye Ngan, “Druid: A System for Demonstrational Rapid User Interface Development”, *Proceedings of UIST ‘90*, Snowbird UT, pp. 167-177.

[34] Sun Microsystems, “The Java Language: A White Paper”, *Sun Microsystems White Paper*, 1994.

[35] S. Takahashi, S. Matsuoka, A. Yonezawa and T. Kamada, “A general framework for bi-directional translation between abstract and pictorial data”, *Proceedings of UIST'91*, Hilton Head, South Carolina, Nov 1991.

[36] Steven H. Tang, and Mark A. Linton, “Blending Structured Graphics and Layout”, *Proceedings of UIST ‘94*, Marina del Rey, CA, Nov. 1994, pp. 167-174.

[37] Brad Vander Zanden, Brad A. Myers, D. Giuse, Pedro Szekely, “Integrating Pointer Variables into One-Way Constraint Models”, *ACM Transactions on Computer Human Interaction*, 1:1, June 1994.

[38] J.E. White, “Telescript Technology: The Foundation for the Electronic Marketplace”, *White Paper. General Magic Inc.*, 1994.

[39] E. Zayas, “Attacking the Process Migration Bottleneck”, *Proc. of 11th ACM Symposium on Operating Systems Principles*, 1987, pp. 13-24.