# A Mediator for Approximate Consistency: Supporting "Good Enough" Materialized Views

Len Seligman
The MITRE Corporation
seligman@mitre.org

Larry Kerschberg
George Mason University
kersch@gmu.edu

**Abstract**

This paper addresses the needs of application designers who would like to tell an automated assistant the following: "Here is a query that defines a view I want to materialize within my application. I need this view to remain *approximately* consistent with the state of the data sources from which the view is derived, in accordance with declaratively specified *staleness* predicates. When the view becomes stale, follow the refresh strategy I specify (e.g., eager, lazy, hybrid). You must do this in heterogeneous environments containing both active and passive data sources."

This paper describes an architecture that realizes this vision. The approach supports materialized, object-based views, called *quasi-views*, defined over shared databases. Quasi-views are refreshed according to the consistency conditions and refresh strategies specified declaratively by application designers. These conditions allow for the deviation of quasi-views from their database counterparts according to well-defined and monitored approximate consistency predicates. A layer of software called a *Mediator for Approximate Consistency* automatically generates the database objects necessary to enforce these consistency conditions, shielding the application developer from the implementation details of consistency maintenance. In addition, it does this for both active and passive (e.g., legacy) data sources.

This paper formalizes quasi-views, presents a declarative quasi-view specification language, and describes an architecture and implementation of a Mediator for Approximate Consistency.

## 1. Introduction

Many data-intensive information systems have databases which are constantly being updated by transactions, by sensor data from satellites, and by value-added processing of data to create information products. In addition, many automated information systems need to:

- transform and cache information from dynamic, shared databases,
- reason about the current state of those data, and
- perform long-running tasks without locking the objects about which they are reasoning, so as to allow concurrent access by other applications.

Many of these applications can tolerate some deviation between the state of their caches and that of the shared databases, as long as this deviation is within specified tolerances. Applications with these characteristics include: (1) coordination of workflow among interconnected aspects of global enterprises (e.g., design, manufacturing, distribution, etc.), (2) network management and fault diagnosis, (3) on-line monitoring of complex environments, such as factories and power plants, (4) automated securities trading, and (4) tactical military planning.

In previous work (Seligman and Kerschberg, 1993a, 1993b, 1995), we described the requirements of such applications and proposed an architecture for addressing those requirements. Since these publications, we have generalized and refined our approach, realized it in prototype software, used it to construct an application, and analyzed its performance. This paper presents these new results.

The approach supports materialized, object-based views, called *quasi-views*, defined over shared databases. Quasi-views are refreshed according to the consistency conditions and refresh strategies specified declaratively by application designers. These conditions allow for the deviation of quasi-views from their database counterparts according to well-defined and monitored approximate consistency predicates.

The approach relies on an intelligent interface to both active and passive data sources that we call a Mediator for Approximate Consistency (MAC). The MAC has several unique features: (1) it permits applications to specify their consistency requirements using a declarative language; (2) it automatically generates the database objects necessary to enforce those consistency requirements, shielding the application developer from the implementation details of consistency maintenance; (3) it does this in heterogeneous environments that include both active and passive (e.g., legacy) data sources; and (4) it provides mechanisms for resolving data heterogeneity issues between client applications and the databases that serve them.

This paper is organized as follows. Section 2 introduces quasi-view objects, which extend quasi-caching to support the transformation of objects before they are cached. In addition, it defines a declarative language for specifying quasi-views, based on a modest extension to SQL, and provides illustrative examples. Section 3 presents an architecture and design for approximate consistency mediators, including an approach to coping with heterogeneous environments. Section 4 describes an implementation of the approach in prototype software, the development of a proof-of-concept application, and a performance analysis. Section 5 compares the MAC with related work. Section 6 presents our conclusions and some directions for future research.

## 2. Quasi-views

This section describes *quasi-views*, which extend quasi-caching (Alonso, et al., 1990) by adding support for: (1) resolving heterogeneity issues between data sources and receivers, and (2) new types of consistency conditions. In addition, this section presents a language for specifying quasi-views and provides illustrative examples.

### 2.1. Quasi-views: Description and Formal Framework

Informally, a quasi-view is a materialized view whose instances are allowed to deviate in controlled ways from the corresponding instances of a similarly defined traditional view. A quasi-view is to a traditional view as a quasi-cache is to a traditional cache in which perfect coherency is maintained. We now define quasi-views more formally, beginning with the concept of a quasi-view server.

**Definition.** A *quasi-view server* QVS is a data source from which quasi-views (defined below) can be derived. Quasi-view servers are typically databases, though they can also be flat files, text-based information retrieval systems, geographic information systems, spreadsheets, or other data sources, as long as they have available query language front-ends or gateways to a local data access language. A quasi-view server QVS may be a centralized or distributed database, or a federation of databases.

Assume that C is a class in a quasi-view server QVS. It has attributes $\{A_1...A_m\}$, each with an associated domain, and may have a set of methods. Each server class C has an extent of instances. An instance of C is referred to as an $o_i$. Each object $o_i$ has a value $v_k$ for each attribute $A_k \in \{A_1...A_m\}$ of C, and that value must be in the domain of $A_k$.

**Definition.** A *quasi-view class* Q is a class in a client application which is defined by a seven-tuple:

$$\mathbf{Q = \langle N, S, A, B, V, \omega, H \rangle}$$

where N is the class name, S is a (possibly empty) set of superclasses, A is a set of attributes, B is a set of behaviors, V is a query language view expression, ω is a set of staleness conditions, and H is a set of implementation hints to the quasi-view server. Some of these require additional clarification:

- **A**, the *attributes* of Q, can include both locally defined attributes, as well as those inherited from any superclasses of Q. A includes attributes whose derivations are specified in the query language view expression V. In addition, there may be attributes in A which are not derived from objects in the server and which are only meaningful within the application.[1]
- **B** is a (possibly empty) set of *behaviors* that can include both locally defined methods, as well as inherited ones.
- **V** is a query language *view expression* against one or more classes in the quasi-view server. This query describes the conditions under which a quasi-copy (defined below) should be created in the client application. In addition, V describes derivations for all derived attributes in A. Each of these derivations is a mapping from a set of attributes of one or more server classes to an attribute $a_i \in$ A. These derivations can use functions which traverse relationship links and can use any methods defined for server classes being accessed. Importantly, the derivations can

[1]These are called "transient slots" in (Paepcke, 1989).

include arbitrary functions, which can be nested as required. As a result, functions can be used as in (Ahmed, et al., 1991) to reconcile both representation and semantic heterogeneity between Q and the server classes from which it is derived.

- $\omega$, the *staleness conditions*, describe the conditions under which instances of Q are to be considered stale—i.e., they no longer meet the specified coherency conditions.[2]
- **H** is a set of *implementation hints*. The most important of these is the *refresh-strategy*, which tells the quasi-view server what to do when a quasi-copy's staleness-conditions have been satisfied. Possible values for refresh-strategy include "eager" (i.e., refresh the quasi-copy immediately), "lazy" (i.e., send only a flag indicating that the quasi-copy is now stale, then refresh when the object is next accessed), "opportunistic" (i.e., "use an eager refresh strategy when the network load permits and a lazy strategy otherwise"), and "eager except <attribute-list>" (i.e., "refresh immediately, except for the listed attributes"). The last option permits an eager strategy to be used for some attributes, while using a lazy strategy for images, video, or other large objects which may be costly to transmit.

  Other possible implementation hints include the priority of messages to the client about this quasi-view class (vis a vis other quasi-views) as well as information about static data that the server need not continually monitor. In certain cases, such hints can have significant performance benefits. The set of allowable implementation hints should be extensible, to allow implementors to provide hints that would be useful in particular environments.

**Definition**. A *quasi-view object* is an object in a client application which is an instance of a quasi-view class. Following Alonso, et al., we sometimes refer to a quasi-view object as a *quasi-copy*.

Quasi-view objects, like the view-objects of Barsalou, et al. (1991), are objects which are constructed from query expressions against a pivot class (or relation) and a set of additional classes (or relations) to which the pivot class is related.[3] Each quasi-view object $o_i'$ of a quasi-view class Q is a *quasi-copy* of some $o_i$—i.e., an instance (or tuple) of the pivot class (or relation) from which Q is derived. In our formulation, unlike that of (Alonso, 1990), a quasi-copy $o_i'$ can be substantially transformed from the $o_i$ from which it is derived.

---

[2]Staleness conditions (i.e., the conditions under which the quasi-view becomes stale) are the complement of coherency conditions (i.e., the conditions under which the quasi-view is still acceptable). We use "staleness" rather than "coherency" conditions, because our users found it more intuitive to specify the conditions under which something ought to happen (e.g., a refresh) rather than those under which coherency remains acceptable.

[3]A pivot relation is a relation on which a view-object is "anchored," constituting its core component. The concept is formally defined in (Barsalou, et al., 1991).

**Example.** A quasi-view server contains the following relation: Stock(name, code, category, current-price, chairman-video). Suppose one wants to create a quasi-view class called Blue-Chip-Stock which is derived from Stock. A possible quasi-view class definition follows:

N = Blue-Chip-Stock
S = {Investment}
A = {name, stock-code, current-price, last-ceo-press-conference, sell-at-price,
    buy-at-price}
B = {Buy, Sell}
V = "Select name, stock-code = code, current-price = Dollars-to-yen(current-price),
    last-ceo-press-conference = Convert-format(chairman-video)
  From Stock
  where category = 'Blue Chip'"
ω = {"When current-price deviates by more than 5% from the currently cached price"}
H = {refresh-strategy = "Eager except {chairman-video}",
  static-fields = {name, stock-code}}

Blue-Chip-Stock has one immediate superclass called "Investment." The attributes of Blue-Chip-Stock are name, stock-code, current-price, last-ceo-press-conference, sell-at-price, and buy-at-price, and its methods are Buy and Sell. The query V indicates that instances of Blue-Chip-Stock should be created for all instances of Stock for which category = "Blue Chip". In addition, the query describes the derivations of name, stock-code, current-price, and last-ceo-press-conference: they get the values of Stock.name, Stock.code, Dollars-to-yen(Stock.current-price), and Convert-format(Stock.chairman-video) respectively, where both Dollars-to-yen and Convert-format are functions defined within the quasi-view server. (Derivations are not provided for sell-at-price and buy-at-price; those are non-persistent attributes of Blue-Chip-Stock which only have meaning within the particular application.)

When the staleness conditions, ω, are satisfied as a result of an update to an instance of Stock, then the corresponding quasi-copy is considered stale. The first specified implementation hint concerns refresh-strategy. Because the refresh-strategy is "Eager except {chairman-video}", stale quasi-copies are refreshed immediately, with the exception of the value of the attribute chairman-video, which is refreshed lazily. The second implementation hint indicates that the values of name and stock-code are static and therefore do not need to be refreshed.

## 2.2. Specifying Quasi-view Classes

This section presents an extension of SQL that can be used to specify quasi-view classes. The proposed language provides a clean separation of the quasi-view definition from optional hints that can be given to the quasi-view server so as to increase performance. An EBNF grammar for the specification language is shown in Figure 1.[4]

---

[4]Space does not permit a detailed language description; such a description can be found in (Seligman, 1995).

```
<quasi-view-def>       ::=  create quasi-view quasi-view-name (<target-list>)
                                [under superclass {, superclass}*]
                                as <select-statement>
                                [with staleness conditions <staleness-conds>]
                                [with hints <hint> {, <hint>}*]
<target-list>          ::=  client-attribute-name {, client-attribute-name}*
<select-statement>     ::=  select [always] <value-exprs>
                                from class-name {, class-name}*
                                [where predicate]
<value-exprs>          ::=  <value-expr> {, <value-expr>}*
<value-expr>           ::=  [server-class-name.]server-attribute-name |
                            constant |
                            function(<value-expr> {, <value-expr>}*)
<staleness-conds>      ::=  any change | never | <staleness-cond> {, <staleness-cond>}*
<staleness-cond>       ::=  any change to server-atttribute-name |
                            {value | percent} server-attribute-name positive-integer |
                            version positive-integer |
                            time positive-integer time-unit |
                            predicate(<value-exprs>) |
                            user-defined-delta
                                predicate (delta(<attribute-list>) [, <value-exprs>])
                                with delta-function function
<hint>                 ::=  refresh-strategy <refresh-strategy> |
                            retract-instances {yes | no} |
                            static-attributes <attribute-list>
<refresh-strategy>     ::=  eager | opportunistic | lazy |
                            eager except <attribute-list> |
                            opportunistic except <attribute-list>
<attribute-list>       ::=  server-attribute-name {, server-attribute-name}*
```

**Figure 1. SQL Extension for Specifying Quasi-view Classes**

Figure 2 illustrates a specification for a quasi-view class called FriendlyTrack, which provides position and other information on military ships and planes from selected allied countries. FriendlyTrack has attributes ID, Velocity, Location, Home-port, Image, Flag, and Force-magnitude, and it has one superclass, InterestingEntity. The “select always” statement means that an instance of FriendlyTrack should be created whenever there is an instance of Track with Flag equal to either “US”, “UK”, or “DE”. Had “always” been omitted, the quasi-view would contain only those instance of Track satisfying the predicate at quasi-view initialization time; the server would not continue to monitor for new Track instances meeting the predicate.

Following the keyword “always” is a list of derivations of attributes of FriendlyTrack. These derivations are illustrated in Figure 3. ID, Image, and Flag are derived trivially from Track.ID, Track.Image, and Track.Flag respectively. Location is derived by applying the function *list* to the arguments Track.Latitude and Track.Longitude. The derivation of Home-port illustrates the resolution of structural heterogeneity; it is derived by applying the *name* function to the value of Track.Home-facility, whose domain consists of instances of the class Facility. More complex forms of heterogeneity could be resolved similarly using arbitrary functions (nested as required) in derivations. No derivation is specified for the attribute force-magnitude, because it is a nonpersistent attribute only meaningful within the application program.

The *staleness conditions* describe the conditions under which the quasi-view no longer meets the user's specified coherency requirements. This occurs when Track.Speed varies by more than 50 percent from the value of Velocity for a corresponding quasi-copy, whenever a cached instance of Track is updated 5 or more times since the last refresh, or whenever Track.Image is updated.

```
create quasi-view FriendlyTrack
    (ID, Velocity, Location, Home-port, Image, Flag, Force-magnitude)
    under InterestingEntity as
    select always ID, Speed, List(Latitude, Longitude), Name(Home-facility), Image,
                Flag
        from Track
        where Flag in ("US", "UK", "DE")
    with staleness-conditions
        percent Speed 50,
        version 5,
        any change to Image,
        user-defined-delta
                > (delta (Latitude, Longitude), 20))
                with delta-function Distance-using-lat-long
    with hints
        refresh-strategy eager except Image
```

**Figure 2. Example Quasi-view Specification**

The last staleness condition in Figure 2 is a *user-defined-delta*, a type of condition which was essential for our prototype applications and which is not discussed in previous work. User-defined-deltas are used to describe data changes in the server that cause quasi-copies to become stale when those conditions depend upon changes to multiple attributes or to non-atomic or non-numeric attributes. The example illustrates how a user-defined-delta can track the magnitude of changes in Location, which is represented as a pair of complex attributes. The interpretation of the last staleness condition is this: a quasi-copy becomes stale when the change in <Latitude, Longitude> exceeds 20, where the delta is measured by invoking the function Distance-using-lat-long with the arguments Latitude, Longitude, Cached-value-for-Latitude, and Cached-value-for-Longitude.

One implementation hint is specified in Figure 2: that the refresh strategy is "eager except Image." This indicates that quasi-copies should be refreshed as soon as they become stale, except for the attribute Image, which is refreshed only when it is accessed.

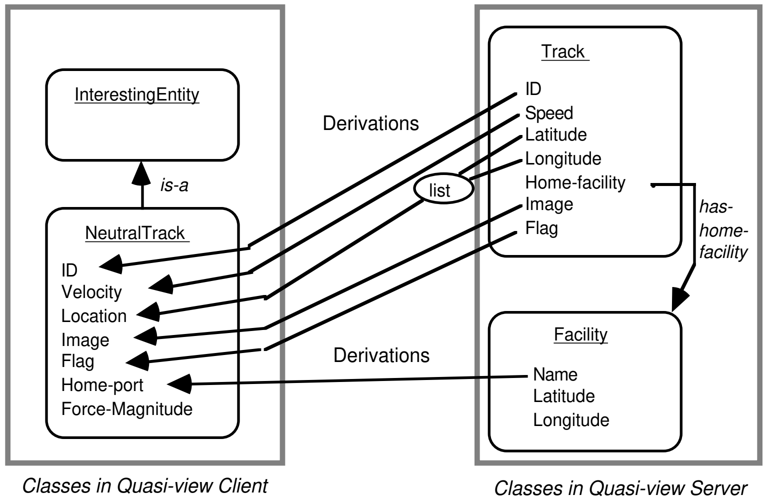


**Figure 3. Attribute Derivations for Example**

## 3. Approximate Consistency Mediation

Central to our approach to consistency management across heterogeneous systems is a *Mediator for Approximate Consistency (MAC)*. An architecture for approximate consistency mediation is presented in (Seligman and Kerschberg, 1993b). This section presents a refinement of that architecture which better supports the needs of heterogeneous systems.

The term *mediator* was defined by Wiederhold (1992a) and refers to software that "simplifies, abstracts, reduces, merges, or explains data" in order to "create information for a higher layer of applications." A mediator for approximate consistency abstracts away most changes to underlying data sources and only reports those updates the application has defined as being "significant." The MAC provides a "consistent enough" view of the component data sources (in terms of the staleness conditions described in a quasi-view specification) for the application to accomplish its tasks.

Figure 4 illustrates our architecture for approximate consistency mediation. The mediator itself consists of three major submodules: the *Translator,* which processes requests from applications to the component data sources, the *Message Handler*, which manages a queue for handling messages from data sources to client applications, and language-specific *application programming interfaces (API)* to the mediator. In addition, the approach relies on *wrappers*, which provide descriptions of the sources, including the capabilities of the database management systems or other software that manage them.

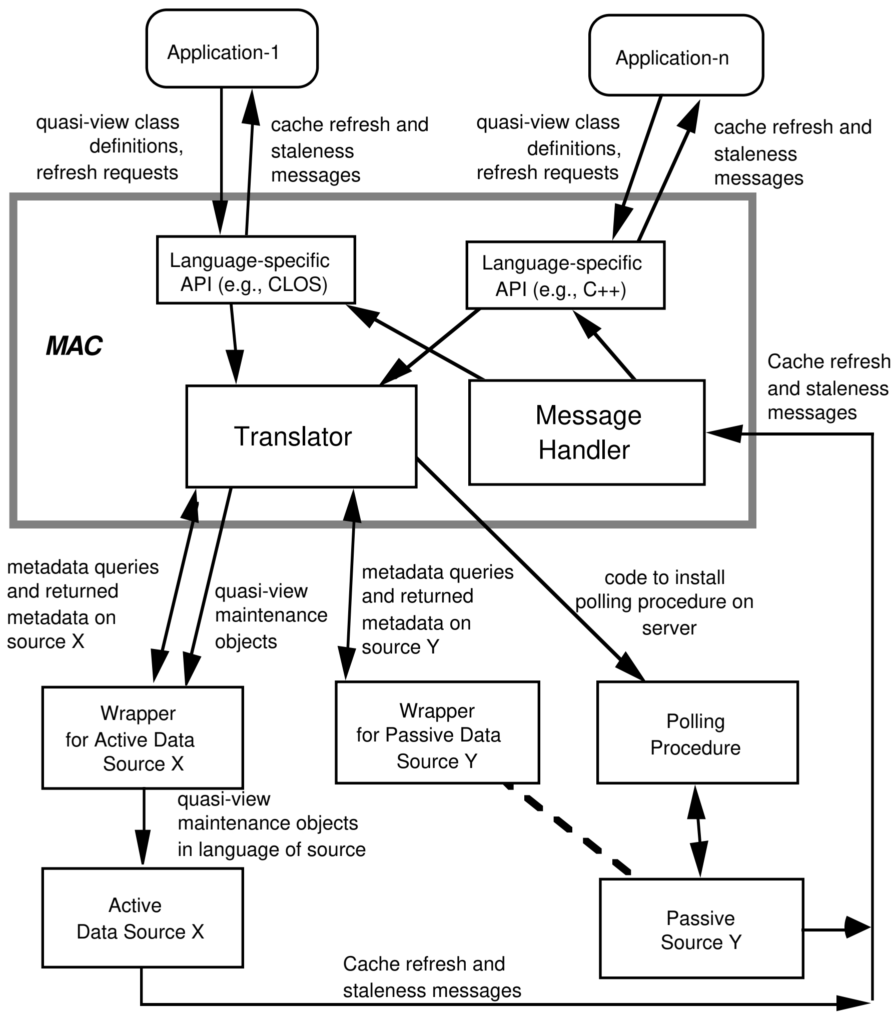


**Figure 4. Architecture for Approximate Consistency Mediation**

An application declaratively specifies its consistency requirements by defining quasi-view classes using either the SQL extension described above or using a language-specific API, such as the one implemented for the Common Lisp Object System (CLOS) (Seligman, 1994). The API converts quasi-view specifications into a canonical form expected by the Translator. Applications also can use library functions provided by the API to modify existing quasi-view class definitions (e.g., their staleness conditions) or to request an immediate refresh of some or all of the instances of specified quasi-view classes.

## 3.1 The Translator

The Translator generates the rules, objects, and/or programs necessary to maintain the specified quasi-view classes. In order to do this in a heterogeneous environment, the Translator queries the wrappers of relevant data sources. Each wrapper returns metadata which describes the active capabilities of the source's data manager (i.e., DBMS, file system, etc.), the privileges granted to the mediator (e.g., does it have permission to create rules on this source?), as well as miscellaneous administrative information.

The Translator uses the metadata returned by the wrappers to determine what objects to generate. There are two main categories of sources for which monitoring objects must be generated: active and passive ones. Active sources are those which support Event-Condition-Action (ECA) rules (Hanson and Widom, 1995), while passive sources are those which do not. Note that active sources can include passive sources (e.g., legacy databases) which are "wrapped" to support rule processing. For example, one could replicate a subset of a passive database in an active one using "asynchronous replication" software such as that offered by Sybase, Oracle, or IBM (Stacey, 1994). The consistency mediator could regard such a source as being active.

For passive data sources, the Translator generates a procedure which polls the source periodically to determine if any quasi-views need to be refreshed. Note that active databases on which the mediator lacks Create Rule privileges must be considered passive by the Translator, unless the source is wrapped as described in the previous paragraph.

For active data sources, the Translator translates declarative quasi-view specifications into the following: (1) queries to be executed immediately, (2) rules for monitoring the future state of the database, and (3) data definition language commands that result in the creation of and updates to staleness conditions as well as other quasi-view maintenance objects in the database. The queries that are to be executed immediately are used to initialize the extension of the quasi-view class. The rules are of three types: *selection-rules,* that are used to monitor the data source for conditions warranting the creation of a new instance of the quasi-view class; *retraction-rules,* that monitor the database for conditions under which a quasi-copy should be purged; and *refresh-rules,* that are used to monitor the database for conditions which cause quasi-copies to be considered stale.

In addition to an initialization query and the rules described above, the Translator must create a number of other objects to help the mediator ensure that the defined coherency conditions are enforced. These objects represent information on the currently defined staleness conditions and the quasi-views they support, what base table (or base class) instances are constrained by which quasi-views, and state information on some of the quasi-copies, particularly those that are constrained by delta conditions (e.g., refresh when attribute $\alpha$ varies by more than 20% from its cached value).

While it would be possible to represent all this information in rules, we recommend against this in order to minimize the number of rules required and to avoid frequent run-time changes to the rule-base. There are two reasons for wanting to do these things. First, there is an autonomy issue. Database administrators are aware that rules have great power. DBAs with whom we have spoken are uncomfortable with having mediators (or users) install and make frequent changes to database rules. Second, unnecessary rule-base changes should be avoided for performance reasons. In many active databases, particularly those that maintain a Rete network or otherwise incrementally compute condition results, adding and deleting rules is computationally expensive.

```
Name:  R_1
Event:  update to track.speed
Condition:  new.ID = “001” AND
                (new.speed ≥ 500 OR
                 new.speed ≤ 200)
Action:
        set rule.condition to
                concat(“new.ID = ‘001’ AND  ”,
                        “(new.speed ≥ ”,
                          compute_new_high_threshhold(new.speed),
                        “  OR  new.speed ≤  ”,
                          compute_new_low_threshhold(new.speed), “)”)
        where rule.name = “R_1”
```

**Figure 5. Example of a Self-modifying Rule**

Based on the above, we offer some design guidelines for approximate consistency mediators, which we have followed in our own implementation. First, we suggest the use of separate objects to represent required information on the state of the quasi-copies, instead of embedding that knowledge in rules. Figure 5 shows an example of a rule that violates this guideline. The problem with this approach is that it requires self-modifying rules, with which few DBAs would be comfortable. Self-modifying rules are required, because every time a quasi-copy is refreshed, the conditions of relevant rules need to be changed to reflect the new thresholds beyond which the quasi-copy again becomes stale. For these reasons, we have removed the quasi-copy state information from monitoring rules and have put it in separate objects, as discussed below.

Another design suggestion is to define the rules that check staleness conditions at the class (not the instance) level. Instance level rules require both more rules and rule base maintenance. E.g., for the example presented in Figure 2, there should be one rule monitoring the database for staleness conditions for all Track instances, not one for each instance for which there exists a quasi-copy.

These guidelines led us to our current design, in which the Translator generates instances of the following classes:

- *Quasi-view*, which contains general information on the quasi-view class.
- *Staleness-condition-definition*, which has subclasses for the different kinds of staleness conditions (e.g., delta, version, user-defined-delta). This class represents all information on a staleness condition that applies across all instances for which quasi-copies exist.
- *Staleness-condition*, which contains information needed to evaluate the conditions described in Staleness-condition-definition objects as they pertain to particular instances. This includes information on the state of the quasi-copies. Subclasses of Staleness-condition include delta-condition, version-condition, and user-defined-delta-condition.

- *Staleness-condition-collection*, which provides a way of grouping all the staleness-condition objects that pertain to a particular database instance.
- *Refresh-rule*, which triggers checking of staleness-conditions. For reasons described above, there is only one refresh-rule for each server class from which quasi-views are derived.

Figure 6 illustrates the objects that are created and maintained by the Translator to support maintenance of approximate consistency. The example chosen is the quasi-view class specification from Figure 2, which has the following staleness conditions:

> **percent** Speed 50,
> **version** 5,
> **any change to** Image,
> **user-defined-delta**
>     > (delta (Latitude, Longitude), 20))
>     **with delta-function** Distance-using-lat-long

The Translator uses this specification of consistency requirements to create the objects shown below the dashed line in Figure 6. Four instances of subclasses of Staleness-condition-definition are created: delta-condition-definition-1, version-condition-definition-1, always-condition-definition-1, and user-defined-delta-condition-definition-1. One rule, refresh-rule-1, is used to determine when the conditions should be tested.

Figure 6 also illustrates what happens when two Track instances, track-1 and track-2 (shown above the solid line), meet the selection conditions for the quasi-view and have quasi-copies created for them. At that time, the objects shown between the dashed and solid lines are created. When the quasi-copies are created, a Staleness-condition-collection is created for each cached instance. Each Staleness-condition-collection contains pointers to instances of subclasses of Staleness-condition. For example, staleness-condition-collection-1 has pointers to its constituent Staleness-conditions: delta-condition-1, version-condition-1, and user-defined-delta-condition-1. These in turn have pointers to the objects that maintain information on the definitions of the refresh conditions—i.e., delta-condition-definition-1, version-condition-definition-1, and user-defined-delta-condition-definition-1. No Staleness-condition object is required for the “any change to Image” condition (i.e., an “always” staleness condition), because this condition type has no need to record information on the state of any quasi-copies.

Now suppose that track-1 has its Speed changed to 26. In that case, refresh-rule-1 would have its event triggered (because of the update to an instance of Track). Because the rule’s condition is always True, the action would be executed. The rule’s action issues a query over instances of Staleness-condition-collection searching for any with a constrained-instance slot that points to track-1. Then, for each Staleness-condition-collection returned by the query (in this case, only staleness-condition-collection-1), the *condition-satisfied* method would be evaluated, which would in turn check delta-condition-1, version-condition-1, always-condition-definition-1, and user-defined-delta-condition-1. Because none of these would return True, the quasi-copy would not be refreshed. The only action that would result is that the *updates-since-last-refresh* slot of version-condition-1 would be incremented.

The fact that the quasi-copy is not refreshed after this update to track-1 is consistent with the goals of quasi-views. Recall that a quasi-view specification declaratively describes what kinds of changes constitute *significant* changes to the application. The application should be insulated from all changes which fail this significance test. In this case, the active database has successfully done this filtering and has done so using objects that the MAC generated automatically from the quasi-view specification.

Now suppose that track-1 has its Latitude changed to “39.8 N” and Longitude changed to “125.1 E”. Again, refresh-rule-1 would have the same initial effect, causing evaluation of the *condition-satisfied* method of staleness-condition-collection-1, which in turn would check the staleness conditions to which it points. When user-defined-delta-condition-1 is checked, the predicate of the corresponding user-defined-delta-condition-definition is evaluated. This time, when the predicate is evaluated, suppose that *distance-using-lat-long* returns 42 (i.e., track-1 has moved 42 miles from the location in the quasi-copy). Since that is greater than 20, the predicate would return True, causing the *condition-satisfied* methods of staleness-condition-collection-1 to return True. Referring to the definition of refresh-rule-1 (shown in Figure 6), one can see that *process-stale-quasi-copy* would be called. This not only refreshes the quasi-copy but also updates user-defined-delta-condition-1 and version-condition-1 with new values for cached-values and updates-since-last-refresh respectively.

Again, this behavior is consistent with the goals of quasi-views. The user-defined-delta-condition-definition that is part of this quasi-view is intended to ensure that only significant changes in location are reported. Because *distance-using-lat-long* reported that the change exceeded the significance threshold specified in the predicate, the quasi-copy is considered stale. The quasi-copy is refreshed immediately with the exception of the attribute Image, in accordance with the stated refresh-strategy. Again, this is accomplished by objects that the MAC has generated automatically from the declarative quasi-view specification.

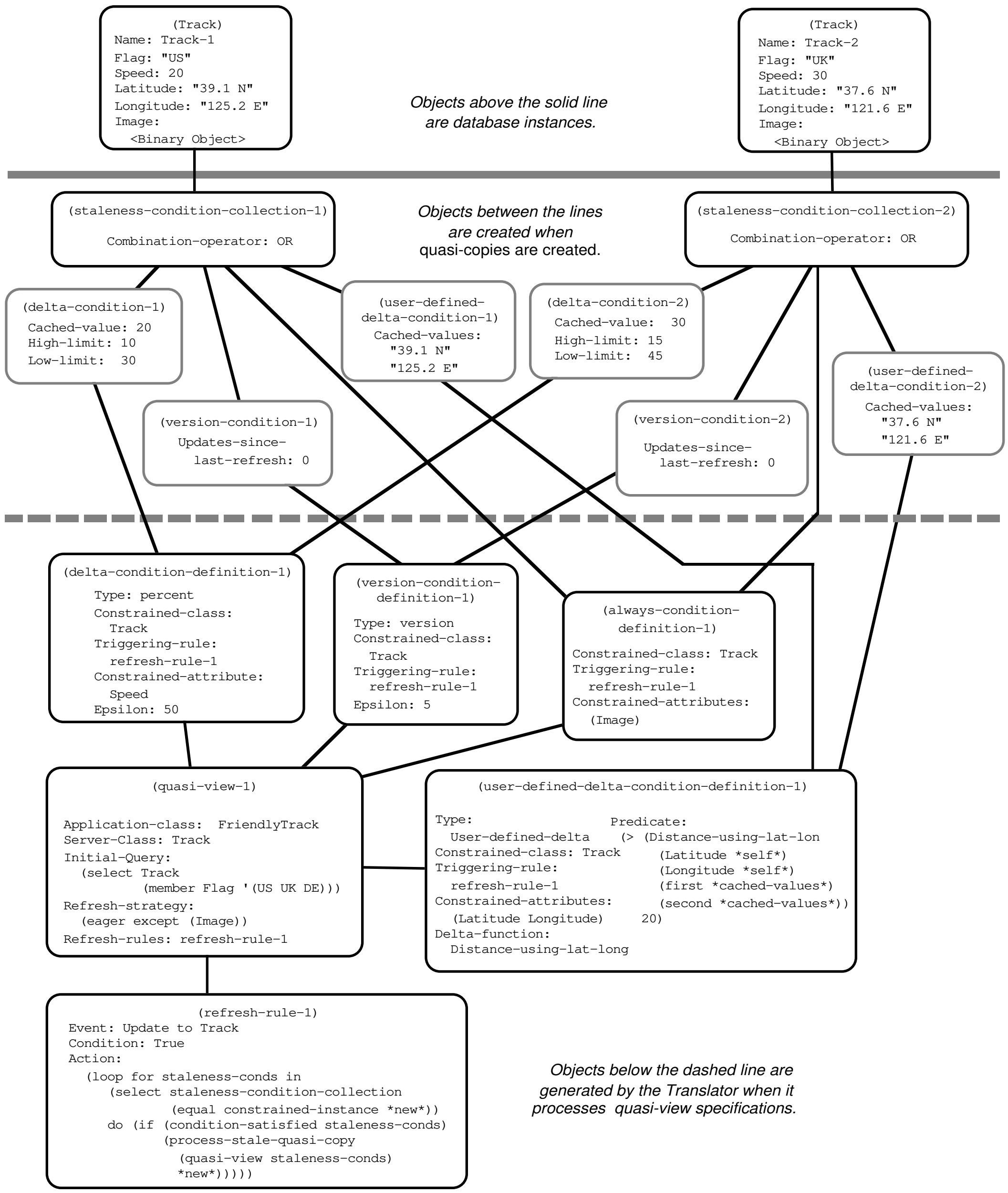


**Figure 6. Objects Generated by the Mediator**

## 3.2 The Message Handler and Application Programming Interface

The previous section described the role of the Translator in generating the objects necessary to monitor both active and passive data sources for changes which should result in creating, deleting, or modifying instances of a quasi-view or in marking them as being stale. Once those objects are generated, there must be a mechanism for notifying the application that these operations should take place. That mechanism is provided by the Message Handler and the API.

The Message Handler receives notification of updates to the quasi-view extension from the data sources and stores those messages in a priority queue until the application is ready to receive them.

The Message Handler accepts both synchronous messages, which are immediate responses to queries forwarded to the database by the Translator, and asynchronous messages, which result from rule firings within active data sources or from generated procedures that poll passive data sources. The asynchronous messages can be either *refresh* messages, which result in creation, deletion, or update of instances in the quasi-view, or *staleness* messages, which mark certain quasi-view objects (or specific attribute values of those objects) as being stale.

The API provides functions to dequeue messages from the Message Handler queue and to map them to operations (e.g., insert, update, and delete) in the client environment. In addition, the API includes support for a lazy refresh strategy. This includes extending the built-in accessor functions for the attributes of a quasi-view class so that they perform object faulting whenever an object (or a specific attribute value) being accessed is marked as stale. For example, the accessor function defined for Stock.chairman-video (from the example in Section 2.1) checks to see if the current value for that attribute is stale. If it is, it issues a refresh request to the translator, which translates the request into a query against the component data source.[5]

### 3.3 Resolving Data Heterogeneity

There are two main types of source/receiver heterogeneity with which applications must contend: infrastructure heterogeneity (e.g., differences in data models, languages, DBMSs) and data heterogeneity. This work addresses some aspects of infrastructure heterogeneity in Section 3.1, particularly differences in the "activeness" of data sources.[6] We now discuss briefly our approach to resolving data heterogeneity, including both representation and semantic heterogeneity.

As described in Section 2, arbitrary functions can be used in the derivations of attribute values for the instances of a quasi-view class. These functions can be used to resolve both representation and semantic heterogeneity between data sources and client applications. One way to do this is for the application developer to specify these mappings explicitly.

A more flexible approach is to use a mediator to generate these mappings, as in the context mediation work of Sciore, Siegel, and Rosenthal (1994). A context mediator can generate a view specification from declarative descriptions of the data receiver (i.e., an application), the data sources, their mappings to a common ontology, and a library of conversion functions. While the MAC could be enhanced to provide this functionality, we prefer to use a more modular approach in which context mediation is separated from consistency management. For example, a context mediator could generate a view that resolves source/receiver heterogeneity. This generated view would be used as the foundation of the quasi-view specification. The application developer would

---

[5]This functionality is most easily provided in languages like CLOS that provide good metaclass support. In these languages, one can create a metaclass for quasi-view class which automatically gives accessor functions for the slots of that class the required object faulting functionality.

[6]Many other aspects of infrastructure heterogeneity (e.g., heterogeneity in data models and languages) are receiving considerable attention in the commercial marketplace (Rosenthal and Seligman, 1994).

need to add only the application's consistency requirements (e.g., staleness conditions and refresh strategy). This kind of flexible combination of mediators is exactly what is envisioned by Wiederhold (1992b).

## 4. Prototype Implementation

The MAC architecture described above has been realized in prototype software. This section describes the prototype and our experiences developing a proof-of-concept application. It closes with a brief discussion of the performance consequences of using the MAC.

### 4.1 Application-independent Portions of the Prototype

The prototype implementation is illustrated in Figure 7. The application-independent parts of the prototype include the MAC, which includes the Translator, the Message Handler, and a Common Lisp Object System (CLOS) application programming interface (API); the ARROW active object database; and an end-user tool for modifying quasi-view definitions.

The MAC prototype implements the functionality described in the previous section—i.e., taking a declarative quasi-view specification and automatically generating the rules and other objects required to properly maintain the quasi-view extension. A detailed description of the techniques for generating these objects appears in (Seligman, 1994).

The current MAC implementation generates these objects for ARROW, an extensible, active object database we implemented by extending Itasca, the commercial implementation of Orion (Kim, 1990). The most notable feature of ARROW, other than support for Event-Condition-Action rules, is its extensibility—its behavior (e.g., its conflict resolution strategy) can be modified easily by making small changes to the methods of a small number of ARROW classes. More detail on ARROW can be found in (Seligman, 1994).

The MAC has been implemented in Common Lisp running on a Sun Sparcstation under the Unix operating system. To date, we have implemented only a CLOS API. However, development of a new API would be a modest effort requiring only the following: (1) a preprocessor to transform quasi-view specifications from an extension of the application language (e.g., a C++ extension) to our canonical quasi-view specification language and (2) code to map canonical refresh messages into statements processible by the application (e.g., *new* in C++).

To define a quasi-view class and its characteristics, one uses the quasi-view specification language described in Section 2.2. Modifications to the specification are done using methods described in (Seligman, 1994), which enable staleness conditions and other aspects of the quasi-view definition to be modified at run-time. While these interfaces are appropriate for programmers, we wanted to demonstrate that tools could be created to assist end-users with defining and modifying quasi-view specifications. In our prototype implementation, we have developed a Staleness Condition Editor, a tool with a graphical user interface that provides assistance with the latter task.

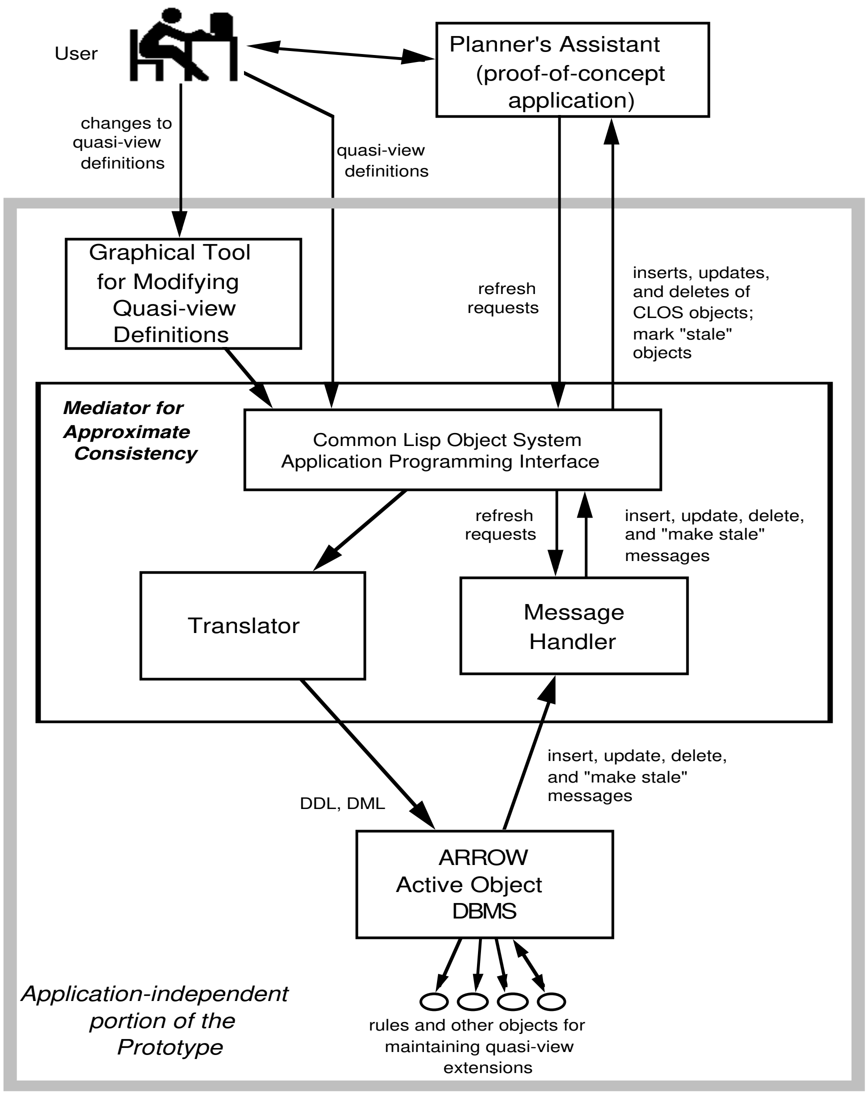


**Figure 7. Prototype System**

Figure 8 shows the basic layout of the Staleness Condition Editor. On the far left is a scrolling list of quasi-view classes defined for a particular client. In the middle is a list of attributes (for the currently selected class) on which staleness conditions can be added and deleted. On the right is a list of the conditions currently defined for that <class, attribute> pair. Finally, there is an English language explanation of the currently selected condition. The condition in Figure 8 means that quasi-copies in this quasi-view should be refreshed whenever the value of wind-speed in the corresponding class in the server changes by more than 15 from the cached value.[7]

Interprocess communication in the prototype is achieved by using Itasca's Remote Lisp application programming interface, which enables two (potentially distributed) processes to communicate using Unix sockets.

[7]The tool assumes the use of an eager refresh strategy.

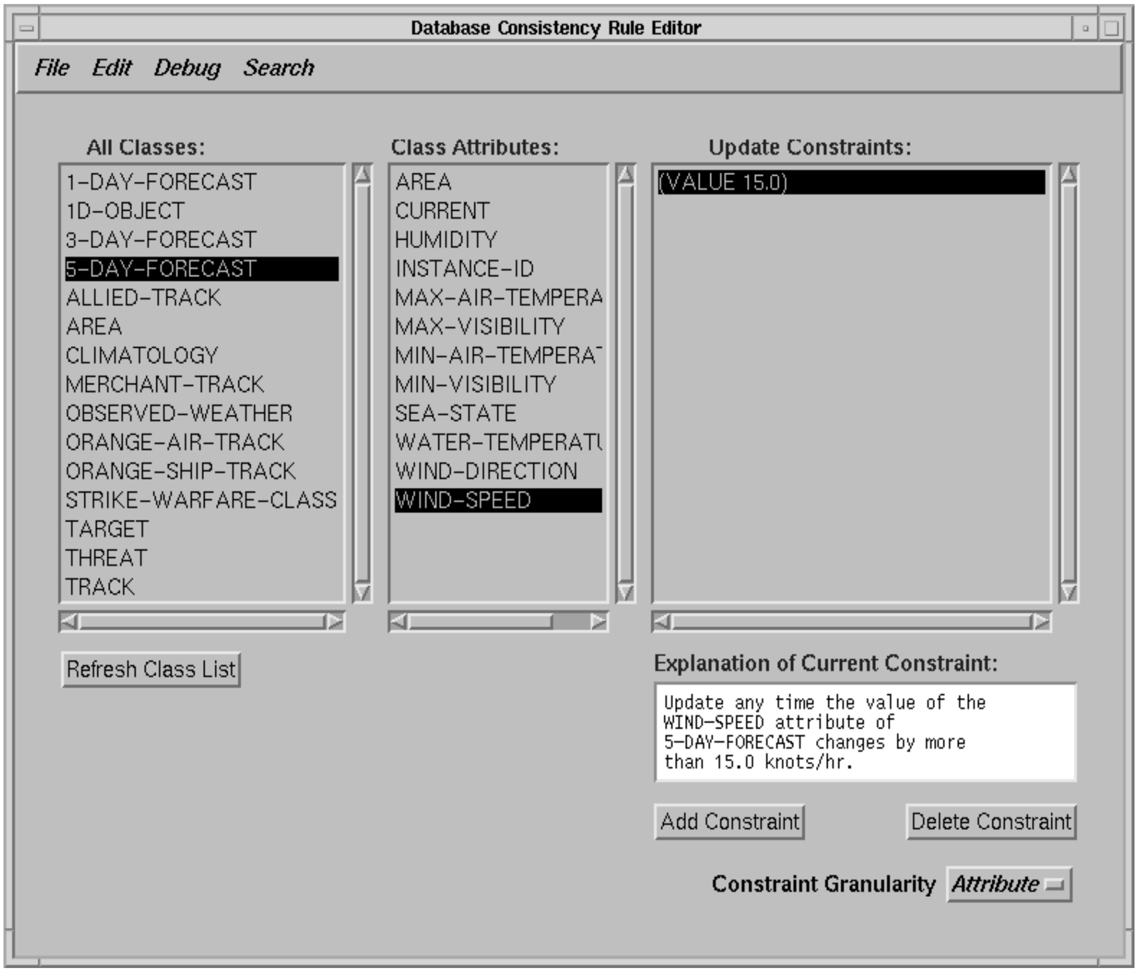


**Figure 8. Staleness Condition Editor**

## 4.2 Proof-of-concept Application

The proof-of-concept application is based on the characteristics of a generic military mission planning system. The database used by the application contains information on the following:

- Tracks—these indicate the presence of some entity at a certain position. The entities could be ships or airplanes and could be friendly, enemy, neutral, or civilian.
- Targets—candidate targets. These have location and description, and are rated according to “value” as well as risk of collateral damage.
- Threats—surface-to-air missiles, artillery, and other threats to friendly assets.
- Climatology—Descriptions of typical weather conditions for an area at a particular time of year.
- Weather reports—These include one, three, and five day forecasts as well as reports of observed weather conditions.

Figure 7 illustrates the application, which supports a human planner who monitors tracks, targets, threats, climatology, and weather reports and uses this information to construct and monitor mission plans. The user describes his data consistency requirements using the quasi-view specification functions provided by the application programming interface to the MAC. Modifications to quasi-view specifications are done with the tool described in the previous section.

Both quasi-view definitions and modifications are sent to the Translator, which generates an initialization query, selection, retraction, and refresh rules, as well as other objects for enforcing the specified conditions. These are all created in the data definition and data manipulation language of ARROW (i.e., the query language of Orion plus extensions for supporting ECA rules). When selection, retraction, and refresh rules fire in ARROW, information on domain instances is sent to the Message Handler. The display program requests messages from the Message Handler via the API, which maps the messages to inserts, updates, and deletes of CLOS objects in the application cache. The display program then displays all relevant application knowledge base updates to the user.

This application has several characteristics that make it well suited to the MAC. First, there is an overwhelming amount of data that could be presented to the user. Users must have the ability to specify what information is significant and have only that information presented to them. It is essential that users be able to describe selection conditions (i.e., the conditions under which they will be made aware of the existence of an entity) and refresh conditions (i.e., the conditions under which they will be notified of updates to a selected entity). Second, users must be able to specify several different types of refresh conditions, as described below. Third, tolerances for the kind and degree of data inconsistency depend on the type of data and on the current situation. Fourth, users must have control over the enforced consistency constraints and must be able to change them with no interruption in service. Finally, this application has another characteristic that makes the MAC especially suitable: The networks have extremely low bandwidth. As a result, minimizing the amount of network traffic is an important goal.[8]

### 4.3 Experiences with the Proof-of-concept Application

This section describes our experiences with the proof-of-concept application. It describes how the MAC supports the definition of data consistency conditions for a realistic application, automatically generates consistency enforcement objects for the defined consistency conditions, and reduces the number of refresh messages from a server to a client application.

In an effort to assess the utility of the MAC, we asked two individuals with considerable expertise in the domain of tactical military mission planning to devise a set of classes that could be used to support a planner and some potentially useful consistency conditions for those classes of

[8]Low bandwidth networks are typical of military environments, but will also become increasingly important in civilian applications with the growth of wireless networks supporting nomadic computing. This sentiment is echoed in a recent paper on nomadic computing which states that "...network bandwidth will remain a major performance bottleneck for system design in the near future" (Alonso and Korth, 1993).

data.[9] The domain experts came up with a set of 11 quasi-view classes, each with an associated selection condition, and with a total of 17 staleness conditions among them.

Using our initial version of the MAC, we were only able to support 7 out of the 17 staleness conditions described by the domain experts. This was because the implementation was limited to supporting the cache coherency conditions described in (Alonso, et al., 1990). However, after expanding the MAC to support arbitrary predicates and user-defined delta conditions, we were able to support all of the specified selection and staleness conditions.

An interesting finding was how useful the user-defined-delta conditions turned out to be in developing the mission planning application. For example, they were essential for tracking the magnitude of changes in the location of objects, as shown in the quasi-view specification in Figure 2.

In addition, user-defined-delta conditions proved to be important even for tracking changes to some atomic attributes, as in the following example:

```
user-defined-delta
      >(delta(wind-direction), 90)
      with delta-function Direction-change-in-degrees
```

In this example, wind-direction is represented in degrees as an integer from 0 to 359, where 0 is North and 180 is South. The reason a simple *value* staleness condition (e.g., whenever the value of wind-direction changes by more than 90) cannot be used is because the change cannot be measured by simple subtraction. For example, a change in wind-direction from 2 to 358 should fail this staleness condition, because it is only a change of 4 degrees, far less than the specified threshold of 90. For this reason, a user-defined delta function, Direction-change-in-degrees, must be used.

Once quasi-views were defined for our application, we next ran the Translator to generate ARROW rules and other objects that enforce the specified consistency conditions. We then sent a stream of updates to the database in order to see that only updates that cause objects to meet the staleness conditions actually result in refreshing a quasi-view. To highlight this, we created an interface with two scrolling windows, shown in Figure 9. The window on the left shows activity against the server database, which in this application is known as the Force Over-the-horizon Track Coordinator (FOTC). The window on the right shows insert, update, and delete messages sent to the Message Handler, in response to selection and staleness conditions being satisfied. Appearing below the message windows are two counters, one showing the number of updates to the server database, while the other one shows the number of updates to quasi-copies caused by staleness conditions being satisfied.

As expected, use of the MAC has resulted in a reduction in the number of refresh messages, compared to approaches that enforce complete consistency. As described above, this is critical to

---

[9]These classes and consistency conditions are not intended to resemble those of any real operational system, however, they do capture the spirit of those systems in terms of the kinds of consistency constraints that seem useful.

the mission planning application because of the low bandwidth networks being employed. In a test run constructed by a domain expert, use of the MAC resulted in 8 updates being propagated to the client out of a total of 138 updates to the database. In other words, 94% of the updates to the database were not considered significant to the application, based on the defined quasi-view specifications. As a result, the updates were not propagated to the application, sparing the users the burden of reviewing data not relevant to their current tasks. Of course, one cannot make generalizations about the magnitude of the reduction in message traffic, because it is entirely dependent upon the staleness conditions that are specified and the nature of updates to the underlying database. Nevertheless, it is encouraging to have obtained this amount of filtering from quasi-view definitions and an update stream that was considered plausible by a domain expert.

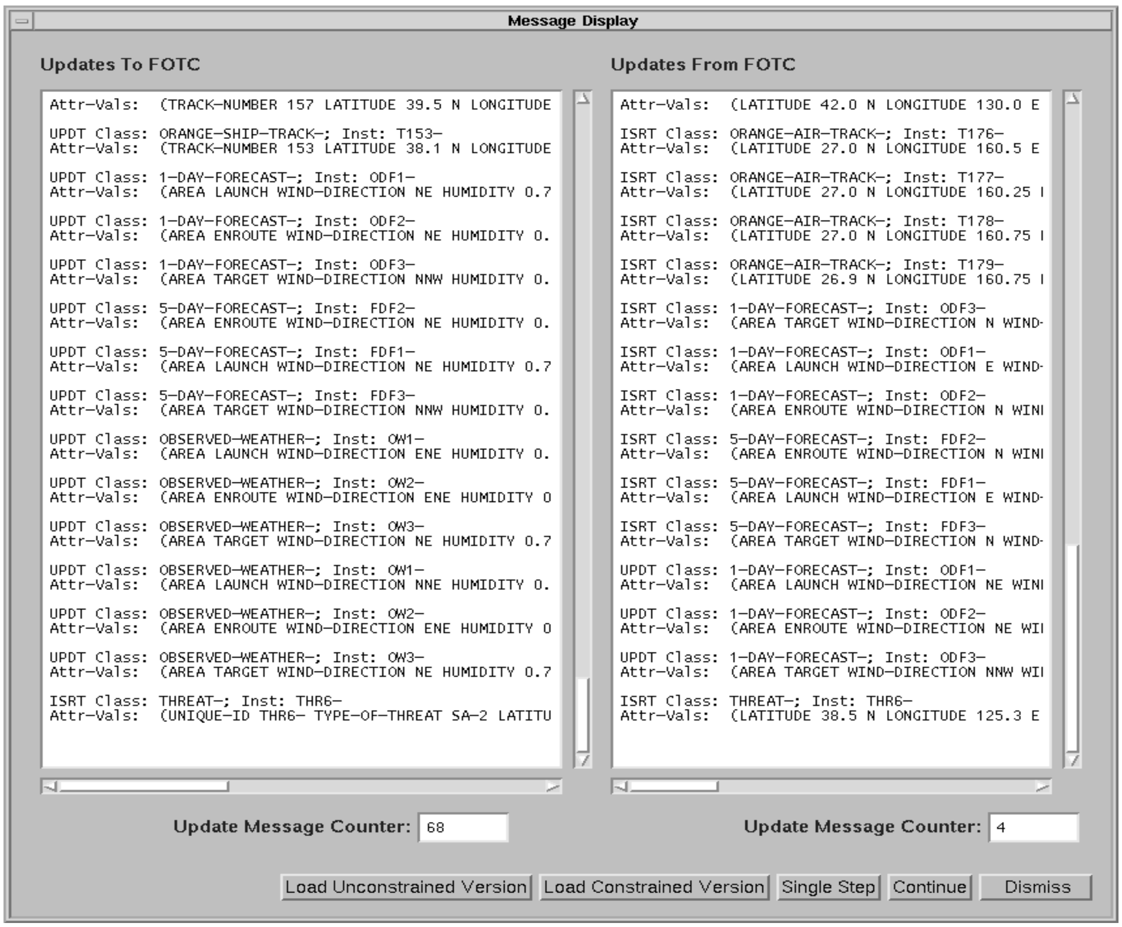


**Figure 9. Updates to Server (on left) and Messages to Client (on right)**

## 4.4 Performance

The MAC results in reduced message traffic as well as reduced processing by client applications, because there are fewer updates for the application to process. As demonstrated in the proof-of-concept application, these savings can be quite significant, depending upon the quasi-views specified and the characteristics of the update stream to server databases. However, the savings are at the expense of increased load on each quasi-view server.

In (Seligman, 1994), we present a detailed analysis of the overhead caused by the MAC. The increased load on each server is caused by the need to check selection, retraction, and staleness conditions. The analysis considers the overhead of checking these conditions when creating, deleting, and updating instances in a quasi-view server.

Our conclusion is that the overhead is primarily a function of Q, the number of quasi-views defined on the class on which updates are being performed. The value of $T_q$, the total number of quasi-copies on any instance of any class in a given server, has only a small effect on the overhead for updates. The assumptions behind the analysis are described in detail in (Seligman, 1994).

Importantly, all of the cost functions are linear, so even if some of our assumptions prove to be unrealistic, only the constants need to be refigured. Also, the overhead per update operation is not affected by the size of the database (e.g., the numbers of tuples). Given the linearity of the cost functions and the irrelevance of database size, the MAC has the potential to scale up to large databases that support multiple client applications.

## 5. Related Work

Quasi-view classes are related to the view-objects of Barsalou, et al. (1991), which are object-based views of relational databases. However, because view-objects are not materialized, they do not support effectively the needs of applications which must cache data, perform long-running analyses of those data, and which need to be informed of changes which the application defines as being "significant." Quasi-view classes support these applications by materializing a view and providing "good enough" (i.e., approximate) consistency between the quasi-copies and the base objects from they are derived.

This work builds upon quasi-caching (Alonso, et al., 1990). Quasi-caches contain quasi-copies, which are client-cached copies of database objects which are allowed to deviate in controlled ways from the primary copies. Their work, like ours, can be said to support approximate consistency of a client cache. Our work extends that work in the following ways. First, we automatically generate rules and other consistency enforcement objects from a declarative specification of consistency requirements. In addition, we have defined an architecture that supports this in heterogeneous environments that include both active and passive sources. Second, we make no special assumptions about the capabilities of our data sources. By contrast, Alsono's work is presented in the context of information retrieval systems with built-in support for quasi-caching. Third, in their approach, for every object o′ in the quasi-cache there is exactly one corresponding object o in the central database, and the representations of o and o′ are identical. Quasi-views generalize quasi-caches to support transformation of the data to be cached (e.g., to support user views or bridge source/receiver heterogeneity). Finally, we have extended delta conditions to support monitoring changes to complex attributes and groups of attributes (e.g., changes to location).

A number of papers address the issue of efficient support for updating materialized views (e.g., Blakeley, et al., 1986; Ceri and Widom, 1991; Hanson, 1987), but these papers do not address techniques for enforcing approximate client cache consistency. The main exceptions to this are Lindsay et al. (1986) and Segev and Park (1989), which describe efficient algorithms for incrementally maintaining database snapshots. These techniques could be used to enforce approximate cache consistency along the temporal dimension, but offer no support for enforcing other kinds of consistency conditions (e.g., delta, version, or user-defined-delta).

A few papers address the issue of interdatabase consistency. Rusinkiewicz et al (1991) have a richer specification language for consistency constraints than we do, but provide no mechanism for enforcing those constraints other than user-defined procedures. Our approach is to automatically generate the required database rules and constraint objects from a declarative specification. Ceri and Widom (1992) provide a declarative specification language for existence and value constraints across components of a multidatabase system and automatically generate active database rules which enforce those constraints. However, they do not provide any mechanism for specifying or maintaining approximate consistency constraints.

## 6. Conclusions

As described in (Seligman and Kerschberg, 1995), many applications need to reason about data which are consistent with the states of dynamic, shared data sources, at least within specified tolerances. These applications require mechanisms for: (1) describing declaratively how consistent their data must be, and (2) for generating consistency enforcement objects from the declarative description of requirements. No previous approach supports this.

This paper has presented an approach to approximate consistency management across distributed, heterogeneous systems. It is the first approach that automatically generates consistency enforcement objects from a declarative specification of application data consistency requirements, where those requirements can include different kinds of *approximate* consistency. It is also the first proposed approach to do this in heterogeneous environments that include both active and passive data sources. We have demonstrated these capabilities in a software prototype and have shown that, given the linearity of the cost functions, the MAC has the potential to scale up to servers with large instance populations supporting many quasi-views.

Another contribution of this work is the introduction and formalization of quasi-views. Quasi-views provide a declarative mechanism for specifying application data consistency requirements. Quasi-views extend quasi-caches by providing: mechanisms to resolve data heterogeneity issues between servers and clients, and a new type of staleness condition, user-defined-deltas, which was essential for our proof-of-concept application. In addition, we have defined a declarative quasi-view specification language, based on a modest extension to SQL. To our knowledge, this is the first formal language defined for specifying approximate consistency requirements.

There are a number of promising areas for future research. First, there is a need to apply these techniques to realistic applications having diverse consistency requirements. Such experimentation could point to a need to support new kinds of approximate consistency predicates. Second, empirical work is needed to replace some of the default parameter values used in our performance analysis with real world numbers. Third, this work could be extended to support a richer set of conditions (e.g., "refresh my cache when the price of some stock goes up by 3% *within one hour*"). Sistla and Wolfson (1995) have developed a rich language for describing temporal triggers. Incorporating such predicates into our mediation architecture would be both useful and challenging.

A final area of research would be to explore the applicability of our approach to data warehousing applications. Typically, warehouses contain static collections of materialized views of multiple heterogeneous data sources (Poe, 1995). The views are static, because there is a requirement that decision support applications not interfere with the performance of databases that support on-line transaction processing (OLTP). However, our techniques have the potential to give warehouse designers a new capability: to specify declaratively changes which are so significant that they should cause an update to the contents of the warehouse. Empirical investigation is required to find out if these techniques are indeed useful for data warehouses and, if so, what changes in our architecture are required to make them so.

## Acknowledgments

This research was partially supported by MITRE Sponsored Research and an ARPA grant, administered by the Office of Naval Research under grant number N0014-92-J-4038. The authors would like to thank Arnie Rosenthal, who provided insightful feedback, especially regarding eager vs. lazy refresh strategies. In addition, we would like to thank Trish Carbone for her creativity in finding applications for this and related research, Eric Peterson for his help with implementation, and Ernie Carbone for his assistance with developing the proof-of-concept application. Finally, the first author would like to thank his management at MITRE for their support, especially Barbara Toohill and Andrea Weiss.

## Bibliography

Ahmed, R., et al. (1991). The Pegasus Heterogeneous Multidatabase System. *Computer*, 24(12).

Alonso, R., Barbara, D., and Garcia-Molina, H. (1990). Data Caching Issues in an Information Retrieval System. *ACM Trans. on Database Systems*, 15(3).

Alonso, R. and Korth, H. (1993). Database Issues in Nomadic Computing. *Proc. of ACM-SIGMOD Int. Conf. on Management of Data*. Washington, DC.

Barsalou, T, Keller, A., Siambela, N., and Wiederhold, G. (1991). Updating Relational Databases through Object-Based Views. *Proc. of ACM-SIGMOD Int. Conf. on Management of Data*. Denver, CO.

Blakeley, J., Larson, P., and Tompa, F. (1986). Efficiently Updating Materialized Views. *Proc. of ACM-SIGMOD Int. Conf. on Management of Data*, Washington, DC.

Ceri, S., and Widom, J. (1991). Deriving Production Rules for Incremental View Maintenance. *Proc. of 17th Int. Conf. on Very Large Data Bases*, Barcelona, Spain.

Ceri, S., and Widom, J. (1992). *Managing Semantic Heterogeneity with Production Rules and Persistent Queues*. IBM Technical Report RJ9064 (80754).

Hanson, E. (1987). A Performance Analysis of View Materializaion Strategies. *Proc. of ACM-SIGMOD Int. Conf. on Management of Data.*

Hanson, E. and Widom, J. (1995). Rule Processing in Active Database Systems. In L. Delcambre (Ed.), *Advances in Databases and Artificial Intelligence, Vol. 1*. Greenwich, CT: JAI Press.

Kim, W., et al. (1990). Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1).

Lindsay, B., Haas, L., Mohan, C., Pirahesh, H., and Wilms, P. (1986). A Snapshot Differential Refresh Algorithm. *Proc. of ACM-SIGMOD Int. Conf. on Management of Data*, Washington, DC.

Poe, V. (1995). Data Warehouse: Architecture is not Infrastructure. *Database Programming and Design*, 8(7).

Rosenthal, A. and Seligman, L (1994). Data Integration in the Large: The Challenge of Reuse. *Proceedings of 20th International Conference on Very Large Data Bases*, industrial track. Santiago, Chile.

Rusinkiewicz, M., Sheth, A., and Karabatis, G. (1991). Specifying Interdatabase Dependencies in a Multidatabase Environment. *Computer*, 24(12).

Sciore, E., Siegel, M., and Rosenthal, A. (1994). Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems. *ACM Transactions on Database Systems*. 19(2).

Segev, A. and Park, J. (1989). Updating Distributed Materialized Views. *IEEE Trans. on Knowledge and Data Engineering*. 1(2).

Seligman, L. (1995). *Quasi-view Specification Using an Extension of SQL.* Working note, available from the author.

Seligman, L. (1994). *A Mediated Approach to Consistency Management Among Distributed, Heterogeneous Information Systems*. Ph.D. thesis, Department of Information Systems and Systems Engineering, George Mason University, Fairfax, VA.

Seligman, L., and Kerschberg, L. (1993a). Knowledge-base/Database Consistency in a Federated Multidatabase Environment. *Proceedings of International Workshop on Research Issues in Database Systems: Interoperability in Multidatabase Systems*. IEEE Computer Society Press.

Seligman, L., and Kerschberg, L. (1993b). An Active Database Approach to Consistency Management in Data- and Knowledge-based Systems. *International Journal of Intelligent and Cooperative Information Systems*, 2(2).

Seligman, L., and Kerschberg, L. (1995). Active Federation: A New Architecture for Integrating AI and Database Systems. In L. Delcambre (Ed.), *Advances in Databases and Artificial Intelligence, Vol. 1*. Greenwich, CT: JAI Press.

Sistla, A.P., and O. Wolfson (1995). Temporal Conditions and Integrity Constraints in Active Database Systems. *Proc. of ACM-SIGMOD Int. Conf. on Management of Data*. San Jose, CA.

Stacey, D. (1994). Replication: DB2, Oracle, or Sybase? *Database Programming and Design*, 7(12).

Wiederhold, G. (1992a). Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3).

Wiederhold, G. (1992b). The Roles of Artificial Intelligence in Information Systems. *Journal of Intelligent Information Systems*, 1(1), Kluwer Academic Publishers.