Middleware Support for Reconciling Client Updates and Data Transcoding

Thomas Phan, George Zorpas, and Rajive Bagrodia

Pervasive Computing Lab
Computer Science Department
University of California, Los Angeles
3809 Boelter Hall
Los Angeles, CA 90095
{phantom, zorpasg, rajive}@cs.ucla.edu

ABSTRACT

In mobile Internet applications, data can be transcoded, updated, and transferred across heterogenous clients. The problem then arises where updates made in the context of an initial transcoding results in content too stringently transcoded for subsequent clients, thereby causing loss of semantic value. We solve this problem by suggesting that the updates themselves can be transformed so that they can be applied directly to the original data instead of to the transcoded data; this approach allows the data to preserve as much semantic value as possible across all heterogeneous clients without unnecessary transcoding artifacts. We define reconciliation rules that can govern the interaction between client updates and transcoding, demonstrate a complete middleware architecture that supports our methodology, and provide two case studies using content-transferring applications. We show that our resulting middleware system executes our reconciliation approach with acceptable latency (under 5 seconds for 200 kbytes of layered content), good scalability, and well-organised modularity.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Communications Applications; D.2 [**Software Engineering**]: Software Architectures

General Terms

Design, Management

Keywords

Mobile computing, reconciliation, transcoding, client updates, data management, middleware

1. INTRODUCTION

Transcoding is a well-known approach to handling heterogeneity at the client end [11] [24]. By taking into consideration a device's hardware, software, and bandwidth limitations, a transcoding mechanism can appropriately adapt the content delivered from an external source to a given client. However, one area that has not been fully explored is the issue of update modifications on transcoded content, a scenario that emerges in mobile Internet applications that share data among clients. In this paper we investigate the problem of how to formalise and reconcile client updates and transcoded data across heterogeneous devices.

These issues are relevant to any environment where data can be transcoded, updated, and transferred between heterogeneous clients. In this paper we study the problem in the context of two compelling application services relevant to mobile computing. Content migration (e.g. [2] [3]) is the term we use for an emerging model that suggests that an application and its content can follow a single user across multiple heterogeneous devices using an application-level suspension, migration, and resumption mechanism. Content sharing (e.g. [15]) can take the form of interactive whiteboards that allow multiple users to convey sketches and other annotations to fellow participants.

The following two scenarios illustrate the problem. For content migration, suppose a user working on a wireless PDA downloads a colour JPEG that has been transcoded to a small, greyscale image due to low bandwidth. After applying some updates to the image, the user migrates his session to his LAN-connected desktop PC. If we follow the intuitive lead that the user should see exactly what he last saw on the PDA, we are left with an undesirable situation: a greyscale image that has been updated is being displayed on the desktop PC's large, full-colour screen. Similarly, consider a content-sharing scenario where users are participating in a whiteboard application session. When a user on a PDA downloads a transcoded image and applies annotations, whiteboard semantics suggest that the other partic-

ipants should see exactly the same content. Even if other users are working on desktop PCs that require no transcoding, the expected whiteboard behaviour leads us to believe that they would be presented with the transcoded, annotated data from the PDA.

In such cases, the transcoding appropriate for one device may be too stringent for other devices. As the data is moved between clients, the transferred content can quickly degenerate monotonically into a form appropriate only for the least common denominator among all the devices. The problem occurs because the update at a low-end device is made in the context of a very lossy transcoding and essentially "locks-down" the content into this heavily transcoded version. Instead, we would like the updated content to be presented as closely as possible to its original form at all subsequent heterogeneous clients regardless of whatever transcoding that may have been applied at a previous client.

In this paper we look to make the following three contributions. First, we formalise the problem in Section 4 and derive a general solution by defining semantic rules in Section 5. We consider the problem in the context of applications where the client updates can be separated from the data on which they operate. These *layered* user modifications allow us to derive the semantic rules. Some important application domains that can use this layered model are medical informatics, computer-aided design/manufacturing (CAD/CAM), and image/photo editing; in our work we use a medical teaching application to demonstrate our ideas. We show that we can leverage layered client updates by *transforming* them to operate on the original data, thereby allowing us to omit unnecessary transcoding artifacts. The end result is that the interaction between updates and transcoding is managed in a predictable and tractable manner. While previous efforts have looked into the problem of managing updates to adapted data (e.g. [4] [7] [8]), to the best of our knowledge ours is the first to reconcile updates and transcoding across heterogeneous devices using a generalisable framework and method of evaluation.

Second, we present a complete middleware service architecture called MoxieProxy in Section 6 that facilitates content migration and content sharing using the semantic rules which we develop. Running on a proxy situated between application servers and client devices, our middleware provides an API and basic services for implementing our semantic reconciliation techniques. The middleware is extensible to support different applications through the use of dynamically downloadable modules that supply the application-specific logic appropriate for particular programs and protocols. Furthermore, our architecture design is scalable across a cluster of workstations and provides transcoding through a pipelined content adaptation mechanism.

Third, we provide two detailed application case studies in Section 7 that demonstrate the combined use of transcoding, client updates, data transfer, and layered operations. iDraw is an image-editing program developed as a follow-up to a medical informatics teaching tool created in-house for radiology professors at our university's hospital. iShare is a multi-client whiteboard application that allows users to convey annotations to multicast participants. Experimental results show that our middleware design executes our reconciliation approach with acceptable latency (under 5 seconds for 200 kbytes of layered content in our sample imaging application), good scalability, and well-organised modularity.

2. RELATED WORK

Building on earlier work, this paper provides semantic rules and a full middleware implementation for content migration/sharing that take into consideration both (1) user-supplied modifications and (2) transcoding of data. To the best of our knowledge, no other research work has provided a generalised solution for this interaction.

2.1 Content migration

Content migration has been suggested as a new paradigm appropriate for the ubiquity of computing devices around us. Generally speaking, in this model applications follow a user across multiple heterogeneous devices. We use the term "content migration" to generalise a class of various application-level implementations and to differentiate this approach from process migration, a brute-force low-level mechanism requiring homogeneous devices [23] [26]. With a content migration capability, an application's session is suspended on a source device, moved to a different target device, and then resumed. The session can also be suspended and then resumed on the same device at a later time. Various approaches to this type of content migration have been seen (e.g. PIMA [3], Follow-Me [31], Multibrowsing [16], One.world [13], and Application Session Handoff [2]), but none have addressed the problem of transcoding and updates. In this paper we will demonstrate our own implementation of a content migration mechanism that follows the high-level reconciliation rules we develop.

2.2 Content sharing

We use the term "content sharing" for the class of collaborative applications, such as Internet whiteboards [15] [25] or desktops [28], that allow multiple users to interactively share and manipulate data. Such a whiteboard provides a GUI for users to draw figures, annotate with text, and erase content over the Internet just as one would with a physical classroom whiteboard. Like content migration, content sharing with a whiteboard is sensitive to device heterogeneity, and the resulting issue of handling updates in the face of needed transcoding is problematic. Prior research in whiteboard applications have not addressed this issue; rather, much work has focused on underlying network support. A seminal reliable multicast design was proposed in [10] using application level framing techniques. A proxy-based architecture was suggested in [7] to support multicast in whiteboards; like our work, they utilise heterogeneous clients but do not address the problem we state here. In this paper we treat the underlying network support abstractly and instead focus on high-level application semantics. To that end, we implemented a simple application-layer multicast using Java RMI (with RPC behaviour) as a communication mechanism to demonstrate our ideas. Because our semantic issues are orthogonal to the underlying communication, our solution can potentially be applied to many whiteboard systems.

2.3 Reconciliation techniques

Mobile computing. The problem of reconciling data after disconnectivity or adaptation has been studied in other mobile computing domains. Within the file system community, reconciling modified files on a mobile host after disconnection from a primary copy has been studied [21], as has prefetching copies in file hoarding [20]. However, neither consider the impact of transcoding effects.
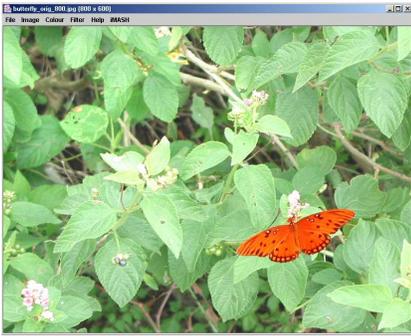
Figure 1: The original 800x600 24-bit colour JPEG image. This represents the object o.

Figure 2: JPEG image from Figure 1 after being transcoded to 400x300 and 4-bit colour to fit a client. This represents f(o).
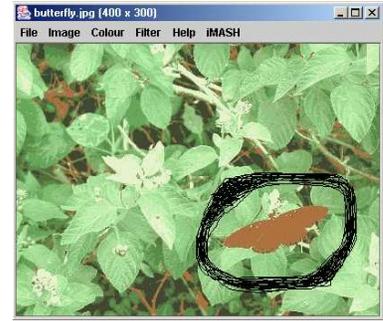

Figure 3: Image after user has circled the butterfly by applying paint strokes onto the transcoded 400x300 photo. This represents g·f(o).

The work in [8] provides an adaptation-aware editing feature that allows users to update transcoded data by suggesting a thorough replication model that takes into consideration the fidelity of subcomponents of content. However, this work does not fully explore how updates on transcoded data should be generally reconciled at the original copy and manifest to other users working on heterogeneous devices. In this regard, reconciliation is done in a type-specific manner (as in our work), but they consider only a subset of the cases we will examine, namely when transcodings and updates are commutative. Our work is fully complementary to their approach; we provide a general framework and evaluative condition for carrying out a reconciliation between transcoding and updates.

Database view updates. Data in a relational database can be filtered by a view and then updated by the user [4], resulting in generally the same problem as ours. The work in [19] paralleled our scenarios and provided key insight by suggesting that any function translating database updates must be created by the database administrator using domain knowledge at the same time that views are defined. Although it is tempting to try to extend other solutions from the database domain into ours, it turns out that the ostensible similarity is a red herring. Other researchers have solved the problem by taking advantage of the reduced set of available view and update operations in the database context by using solutions too database-specific to be generalised. In early work, applying a "constant complement" to database views was suggested [4], but this solution is too dependent on database semantics. Likewise, other approaches included SQL relational algebra [5] and constraint satisfaction techniques [30]. Finally, the SQL2 standard itself already constrains the set of updates available on views.

3. BACKGROUND

3.1 Data transcoding

Transcoding [11], also known as distillation or content adaptation, is the process of changing data, usually with loss of quality, to appropriately fit the limitations of heterogeneous client devices or network bandwidth. Data can be transcoded across different types (e.g. PDF to text) or within the same type (e.g. reduction of colour depth or resolution of JPEG images). This functionality is typically performed at a proxy placed between the application server and the client. Transcoding has been well-researched (e.g. [27] [6] [14] [24]) and shall not be discussed here in detail.

Here we distinguish transcoded data sent over the network with a notation we will use extensively. Let C_1, C_2, ... C_n be a set of n client devices among which content can move. From any client, the user can request a data object o from a server. A middleware proxy then performs the transcoding, or filtering, function f_i on the data object o appropriate to a given client C_i. The operation f may be null if the client does not require any transcoding.

To demonstrate an application of this discussion, in Figure 1 we show a displayed 800x600 24-bit colour JPEG image downloaded from a server using an Internet-enabled imaging program called iDraw, which we will discuss in the next subsection. Although the program is intended for the domain of medicine, here we chose a butterfly image to demonstrate our ideas in the clearest visual manner. The image shown represents a data object o prior to being transcoded at the proxy. In Figure 2 we show the image after going through our transcoder, which we will discuss in detail later. Here, our transcoder, using a static client profile, has distilled the image to fit a 400x300 display with 4-bit colour. The transcoded image thus represents $f(o)$. We state that the absolute quality of the resultant transcoded image is not important in this paper; rather, just the fact that an image was transcoded is what matters. (For completeness though, we note that although there may not appear to be much difference between the two colour photos if they are printed as black-and-white on paper, much detail has been lost. For instance, the butterfly and leaves have lost much interior detail. Furthermore, the figures are not displayed to scale; Figure 1 is exactly twice the size of Figure 2.)

3.2 Client updates via layered operations

Once a data object is delivered to the client, the application accepting this data may provide the user with a means to update it. In this paper we focus on applications whose data modifications can be separated from the objects upon which they operate. We say that in these types of applications, the data modifications are *layered*. The best examples are instructional tools (e.g. used in medical informatics), CAD/CAM (used to design and annotate complex systems), and image-editing programs (e.g. Adobe Photoshop, Jasc Paint Shop Pro, or GNU Gimp). In these applications, user

operations can be naturally represented as a layer that is applied to a substrate, which can be either the original image or one or more other layered operations on the original.

Layered operations are important because they allow us to separate an operation from its operand and thereby provide the flexibility to establish the semantic rules we will define in the next section. Furthermore, because such operations can be decoupled from its operand, we can take advantage of this attribute as a performance optimisation when uploading data from a client to a supporting proxy [22]. As we will also show later, decoupling the operations additionally allows us to transform the operation itself [9].

To facilitate further discussion, we show case studies using two programs. iDraw is an Internet-connected image-editing program that follows up a real-world instructional tool developed in-house by a medical informatics team at our university to support radiology professors. With this program, clinicians download medical images (e.g. magnetic resonance images) and annotate them accordingly. iShare is a multi-user whiteboard application in a similar vein. Both are stand-alone Java programs subclassed from a common GUI drawing application baseclass. These programs can open image files loaded from the local disk or downloaded over the network from an application/WWW server. The images can then be edited using a variety of tools to apply, for instance, brush strokes or typed text; iDraw also provides imaging filters such as sharpening and edge detection. Furthermore, images can be uploaded back to the application server. Since these two applications' front-end functionality is so similar, we will use iDraw to visually drive our problem statement in this paper. Detailed case studies of these applications and our middleware will be provided later.

Layered user operations can be represented using the same style of notation we introduced earlier. Let object $\mathbf{f_i}(\mathbf{o})$ be the transcoded data received by client $\mathbf{C_i}$ from over the network, as discussed before. We say that a user operation $\mathbf{g_i}$ can be applied on said data object at that client. The operation $\mathbf{g_i}$ can consist of $\mathbf{m}$ different layered operations $\mathbf{g_{i1}} \ldots \mathbf{g_{im}}$ concatenated together. The resultant object is $\mathbf{g_i} \cdot \mathbf{f_i}(\mathbf{o})$, which represents, reading from right to left, the original object $\mathbf{o}$, transcoded for client $\mathbf{C_i}$ by the filter $\mathbf{f_i}$, and modified by the user with operation $\mathbf{g_i}$.

Accordingly, the iDraw content is defined along the lines of layers. An opened image, either from a local file or retrieved over the network, is treated as the base layer. Manipulations of the image are abstracted into commands that are serially applied as layers on top of the image. When the user performs a modification, the action is carried out while an XML-based representation of this action is pushed onto an image-manipulation stack. In Figure 3 we show a number of paint strokes circling the butterfly in the transcoded photo; this represents the object $\mathbf{g} \cdot \mathbf{f}(\mathbf{o})$. Each paint stroke is represented as a command defining two cartesian endpoints and a colour. Text annotations are similarly represented as coordinates, text, and font data. Additionally, image filters like Gaussian blurring, sharpening, edge-detection, and colour inversion are also represented using XML. The application's content thus comprises all these preserved commands encoded in XML along with the base image.

4. PROBLEM STATEMENT

At the heart of this paper is the definition of the semantics governing transcoding and updates in an integrated,
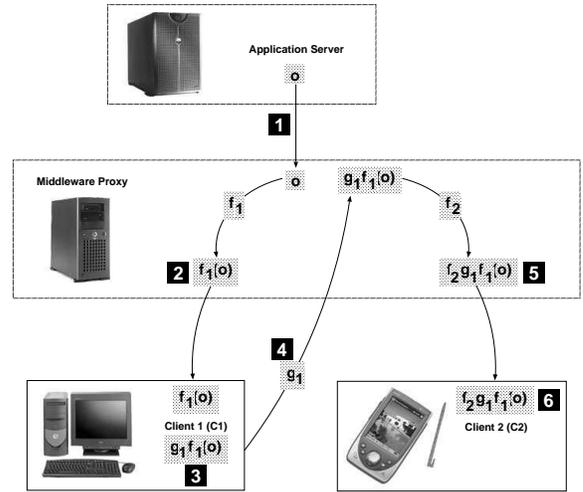

Figure 4: Content transfer events.

mutually-aware, and predictable manner. The problem is relevant to mobile applications that receive transcoded content, allow users to update it, and transfer the resultant data, all across heterogeneous devices.

As mentioned earlier, the two application services we consider, content migration and content sharing, have these characteristics in common. One difference between these two is that in content migration, the data transfer between clients is a state migration for a single user across multiple devices. For content sharing, the data transfer is a sharing, or dissemination, between multiple users across multiple devices. The implementations can be different: the content migration can be provided with any application-layer protocol (we use Sun's RMI communication library), whereas a content-sharing whiteboard typically uses a form of multicast (we use our own application-layer multicast protocol). However, at the high abstract level they all serve the basic purpose of moving the user's content between heterogeneous devices via an intermediary that provides added services. We thus group the data movement stage common to both these scenarios under the umbrella term *content transfer*.

4.1 The problem

The content transfer capability must take into account four important interrelated factors. Multiple cycles of these factors can occur depending on the user's actions.

To provide detail, we present here an example showing a sequence of steps during a content transfer that involves both transcoding and updates. In our discussion we model content transfer using a generalised architecture where a proxy is positioned between a client and an application server to implement transcoding and other services such as caching and service discovery. Such a tiered design is commonly used; we describe our own proxy-based middleware architecture in full detail in Section 6.

In our example in Figure 4, a user initially starts working on a client $\mathbf{C_1}$ and then requests a data object $\mathbf{o}$ from the application server. A middleware proxy supporting the application intercepts the request and asks for the data from the application server on behalf of the client. The server replies with the object $\mathbf{o}$ by sending the object to the proxy, which immediately caches the object, as shown in step 1 of the figure. The following steps then transpire:

- **Initial (pre-transfer) transcoding**. Data received from an external source is transcoded to fit the user's device C_1. This transcoded data is $f_1(o)$ and corresponds to step 2 in the figure.

- **User updates**. At each client device the user has the opportunity to make changes to the received content through whatever user interface is provided by the application. The updated data is $g \cdot f(o)$ and corresponds to step 3.

  At this point the user wishes to perform a content transfer to a target machine C_2 due to either content migration or content sharing. Here we leverage the fact that the user modification g can be decoupled from the data on which it operates as a separated layer. With that in mind, upon transfer only g_1 (rather than $g_1 \cdot f_1(o)$) is sent back to the proxy to be applied remotely there; this transmission is step 4.

- **Subsequent (post-transfer) transcoding**. The content is appropriately transcoded upon transfer to a subsequent device. This transcoded data is $f_2 \cdot g_1 \cdot f_1(o)$ and corresponds to step 5.

- **Preservation of semantic value**. When the content is resumed, the data as well as the user's changes should retain as much semantic value as possible; that is, after the content transfer, the content should convey almost exactly the same information as before. Note that this may not be the case in our example when $f_2 \cdot g_1 \cdot f_1(o)$ is finally delivered to C_2, as shown in step 6. (For a content-sharing application across multiple clients, C_3 would get $f_3 \cdot g_1 \cdot f_1(o)$, C_4 would get $f_4 \cdot g_1 \cdot f_1(o)$, etc.) The delivery of $f_2 \cdot g_1 \cdot f_1(o)$ to the target client is naive and ultimately incorrect.

The sequence $f_2 \cdot g_1 \cdot f_1(o)$ would be correct if the transcoding function f_1 always produces less lossy results than transcoding function f_2. However, C_1 could be a low-end, low-bandwidth wireless PDA while C_2 could be a high-end, high-bandwidth desktop PC. Here, f_1 would be more lossy than f_2.

In this latter case, a problem arises due to the user's updates after an initial transcoding. When the user performs an update on transcoded content, he is explicitly acting on the transcoded data whereas implicitly what he really intends is to update the original data; this intent is an important assumption in our argument.

Furthermore, updating the data in the context of the original transcoding results in content too stringently transcoded for subsequent clients, resulting in loss of semantic value. The result is that as the data is transferred across heterogeneous devices and updated at clients, the content will monotonically degenerate into a lossy version appropriate only for that device that required the most transcoding. The important issue we address is that we ideally want to allow *user updates to be applied to the original, nontranscoded data.*

It follows that the user's g updates should be applied directly to o instead of to the transcoded $f(o)$. The key then is to remove the initial f transcoding. We thus say that applying g directly onto o preserves more of the original data's semantic meaning by omitting the initial f transcoding. In this case the user's updates can be considered applied both implicitly and explicitly to the original data, and the monotonic degradation of quality will not occur.

Referring back to the figure, we thus ideally want C_2 to receive, after adding the f_2 transcoding, $f_2 \cdot g_1(o)$ instead of $f_2 \cdot g_1 \cdot f_1(o)$. It can further be seen that f_2 is irrelevant to the argument: regardless of whatever f_2 may be, the important point is the removal of the initial f_1 transcoding so that g_1 can be applied directly to o.

4.2 The importance of transcoding independence

Unfortunately, this solution of omitting the initial transcoding and applying operations on the original data is problematic. Generally, a user operation may be tightly associated with a specific transcoding that occurred on the data. We introduce two terms that characterise this behaviour:

- **Transcoding independence**. A client update operation is transcoding-independent of a specific transcoding if the operation can be applied either on a transcoded datum or on the original, nontranscoded datum.

- **Transcoding dependence**. A client update operation is transcoding-dependent on a specific transcoding if it can only be applied on the transcoded version of a datum.

Transcoding-independent operations are advantageous in our context because they allow the initial transcoding step to be omitted, thereby allowing user updates to be applied directly on the original data. Upon transfer to another device, the content is then limited only by the post-transfer transcoding. Thus, our desired factor of preservation of semantic value can be attained. Transcoding-independent user operations may require careful thought to derive: if a user operation is transcoding-dependent, it may be possible to transform it into a transcoding-independent operation using a methodology we will explain later. If a transcoding-dependent operation cannot be transformed into a transcoding-independent form, then the user must accept that his data and interactive updates have been limited by both the pre-transfer and the post-transfer transcodings. In the following section we will develop precise rules to apply these definitions and to determine when we can transform a transcoding-dependent operation into a transcoding-independent one.

5. A RECONCILIATORY SOLUTION

Using the previous notation, we observe that the reconciliation problem can be addressed if we apply the user operation g directly to o. We must note that in general across all applications, this is not possible, but in this section we specify a condition which can be used to determine if it is possible. We want to transform a transcoding-dependent update g performed on $f(o)$ at the client into a transcoding-independent update G that can be directly applied to o at the proxy.

5.1 An evaluative condition

Referring back to Figure 4, upon naive post-transfer resumption, we would receive $f_2 \cdot g_1 \cdot f_1(o)$. At this point we make two simplifications without loss of accuracy. First, we drop the subscript for g_1 because we will focus on only one user update. Second, since as we have said that the second transcoding f_2 is not important, we will replace it with the

same $\mathbf{f_1}$; this corresponds to a content migration or content sharing back to the same device, a base case that should always hold true. After transfer, the content should therefore be $\mathbf{f_1 \cdot g \cdot f_1(o)}$.

As stated previously, to ideally reduce the number of transcodings, it is the first transcoding operation that must be eliminated. We therefore want to change the post-transfer series of layerings from $\mathbf{f_1 \cdot g \cdot f_1(o)}$ to $\mathbf{f_1 \cdot G(o)}$, where $\mathbf{G}$ is the operation $\mathbf{g}$ that has been transformed to operate directly on $\mathbf{o}$. We thus ideally want the following condition to be true:

$$\mathbf{g \cdot f_1(o) = f_1 \cdot G(o)}$$

In general, the exact equality may be difficult to achieve. Intuitively, it implies an ideal scenario where the post-transfer state is no worse in quality that the pre-transfer state. The left-hand side represents the user-modified, transcoded object that was last seen at the client before transfer. The right-hand side represents our idealised situation after the object is transferred to the client: a single necessary $\mathbf{f_1}$ takes place on a modified object $\mathbf{G(o)}$ *without* a prior transcoding. If the condition $\mathbf{g \cdot f_1(o) = f_1 \cdot G(o)}$ is satisfactorily met, it implies that the transcoding-dependent $\mathbf{g}$ can be applied to $\mathbf{o}$ directly through its transcoding-independent form $\mathbf{G}$.

5.2 Deriving transcoding-independent G

$\mathbf{G}$ must be obtained from $\mathbf{g}$ somehow. We first consider three simple cases. First, suppose $\mathbf{g}$ is already transcoding-independent of any $\mathbf{f}$; then $\mathbf{f_1}$ and $\mathbf{g}$ are mutually independent and thus commutative. If this were the case, then $\mathbf{g \cdot f_1(o) = f_1 \cdot g(o)}$ by definition and $\mathbf{G(o)}$ is trivially $\mathbf{g(o)}$. Second, if all the transcodings $\mathbf{f}$ are always null, then clearly $\mathbf{g \cdot f_1(o) = f_1 \cdot g \cdot f_1(o)}$ reduces trivially to $\mathbf{g(o) = g(o)}$. However, in the interesting contexts we are researching, $\mathbf{f}$ will be non-null. Finally, for an $\mathbf{o}$ created or loaded locally at the client, the case trivially reduces to where the pre-transfer transcoding is null and no $\mathbf{G}$ is needed; the subsequent client then receives $\mathbf{f_1 \cdot g(o)}$ after transfer, as expected.

We now consider a more interesting case. Suppose that for a given $\mathbf{g}$, there does exist a $\mathbf{G}$ such that $\mathbf{g \cdot f_1(o) = f_1 \cdot G(o)}$ is true. In that case, after rearranging, $\mathbf{G(o) = f_1^{-1} \cdot g \cdot f_1(o)}$. However, in our context any $\mathbf{f}$ represents a lossy transcoding, which by definition is a many-to-few relationship. For instance, reducing colour depth clearly maps from many to few colours. Determining the inverse function $\mathbf{f^{-1}}$ of a many-to-few relationship is not possible because the inverse few-to-many relationship cannot be determined. As such, it is not possible, without additional information, to determine an appropriate re-colourisation to be the inverse of a colour depth reduction. Thus, $\mathbf{G}$ is hard to obtain automatically from solely $\mathbf{f_1}$ and $\mathbf{g}$. We therefore rely on a predefined mapping between $\mathbf{g}$ and $\mathbf{G}$ using per-application heuristics. Such an approach can be represented by a transform function $\mathbf{T}$ that maps $\mathbf{g}$ to $\mathbf{G}$ such that $\mathbf{G = T(f_1, g)}$.

$\mathbf{T}$ represents a heuristic that must be derived from $\mathbf{f_1}$ and $\mathbf{g}$ and therefore has no closed form. Intuitively, one can think of $\mathbf{T}$ as "cancelling out" the effect of a transcoding function $\mathbf{f_1}$, thereby allowing a user operation $\mathbf{g}$ to be applied on $\mathbf{o}$. In general, a $\mathbf{T}$ does not exist for arbitrary $\mathbf{f}$ and $\mathbf{g}$. If a $\mathbf{T}$ *does* exist (that is, if $\mathbf{G}$ can indeed be applied to $\mathbf{o}$), then we say that the transcoding-dependent $\mathbf{g}$ operation can be transformed into a transcoding-independent $\mathbf{G}$. The end result of deriving $\mathbf{G}$ and applying it to $\mathbf{o}$ without a

prior transcoding is that it preserves more semantic value upon content transfer. If a $\mathbf{T}$ does *not* exist (that is, if no transcoding-independent $\mathbf{G}$ can be found), then we must treat $\mathbf{g \cdot f(o)}$ as a new object at the proxy with resulting loss of semantic value upon transfer.

5.3 Programming methodology

In the previous subsection we identified the importance of deriving transcoding-independent functions and the condition which can be used to evaluate an implementation. Here we provide a generalised framework and guidelines for carrying out this approach. An application with layered operations can be modified to use a content transfer with our reconciliation rules by following a simple programming methodology involving a collaboration between the application programmer and the middleware proxy service provider. This approach can be summarised in the following steps:

1. The middleware proxy service provider enumerates the available $\mathbf{f}$ transcoding operations associated with a given data type. For common data types (e.g. JPEG), transcodings can leverage off-the-shelf code.

2. Collaborating with the service provider, the application programmer combines his knowledge of the $\mathbf{g}$ user update operations available in his program with the given set of transcoding operations. This effort produces the $\mathbf{T}$ transform functions.

3. Using the transform functions, the programmer creates the resultant transcoding-independent $\mathbf{G}$ user operations that can be applied to nontranscoded data.

4. The application programmer evaluates the result to see if the pre-transfer and post-transfer presentations are acceptably similar.

During the course of carrying out this methodology, it is helpful to create a table that allows the programmer to evaluate the conditions necessary for reconciling the updates and transcodings. In Table 1 we show such a table that lists the ($\mathbf{f}$, $\mathbf{g}$, $\mathbf{T}$) tuples that must be evaluated. The table is structured following the above design steps and is built from the point of view of the application programmer. The table shows that the $\mathbf{f}$ transcoding is provided to him (by the service provider), the $\mathbf{g}$ user update operation is already known to him, and the $\mathbf{T}$ transformation must be derived (if it exists) to create the resultant $\mathbf{G}$ transcoding-independent operation. The programmer then evaluates the pre- and post-transfer conditions to see if they are acceptable.

We believe this methodology that the provider and programmer can collaboratively follow is reasonable and tractable. We successfully utilised this approach during the development of iDraw and iShare with the support of our middleware. The following examples illustrate the methodology.

5.4 Example: imaging application

Figure 5 shows a naive attempt to apply a user operation $\mathbf{g}$ directly to $\mathbf{o}$. Here we again show paint strokes around the butterfly as we showed in Figure 3. In that previous figure, the operation $\mathbf{g}$ had been applied to $\mathbf{f(o)}$ at the client, thus forming $\mathbf{g \cdot f(o)}$, which represented paint circles within an image that had been transcoded via colour depth and resolution reduction. As noted earlier, the layered $\mathbf{g}$ paint

	f (*provided*)	**g** (*already known*)	**T** (*derived*)	Evaluation (**g·f**(**o**) = **f·G**(**o**) ?)
1	image resolution reduction	paint strokes	scale strokes up	Are the strokes applied in the correct position?
2	image resolution reduction	text annotation	scale text up	Is the text at the correct position?
3	image colour reduction	paint strokes	N/A; apply default colour	Does the default colour convey the same information?
4	image colour reduction	image sharpening	N/A	possibly commutative; depends on implementation of sharpening algorithm
5	PDF-to-text	cut and paste in text	cut and paste PDF text	Are the edits applied at the correct position in the PDF?
6	PDF-to-text	underline in text	underline PDF text	Is the correct text underlined in PDF?
7	speech-to-text	cut and paste in text	cut and paste speech segments	Are the edits applied at the correct position in the speech?
8	speech-to-text	underline in text	raise speech volume	Is the appropriate text/voice-segment underlined/raised?

Table 1: Evaluation table used during the design methodology from the point of view of the application programmer. We used a similar table during the development of iDraw. The last four entries are potential evaluations for PDF-to-text and speech-to-text transcodings we will investigate in the future.

Figure 5: The transcoded 400x300 image after the user's brush strokes, **g**, have been naively applied to the object **o**. Note the misapplied circle above-left of the butterfly. This represents **f·g**(**o**).

Figure 6: Image after user's paint strokes, **g**, were transformed intelligently to **G** (via the transform **T**) and applied to original 800x600 image. This represents **G**(**o**).
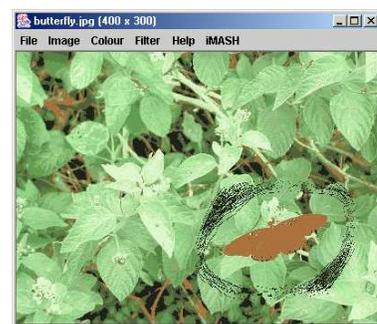

Figure 7: The transcoded version of the image from Figure 6, now reduced to 400x300 and 4-bit colour. This represents **f·G**(**o**).

operations are represented using XML tags containing cartesian linepoints and a colour to be applied as single strokes to a target. Figure 5 shows that applying such a **g** operation directly to **o** without first transforming it to an appropriate form **G** is incorrect. Here, the **g** operations' target is not the transcoded **f**(**o**) but rather the original image **o** without a change in colour depth or, more importantly, size. The original **o** has a resolution of 800x600, whereas the transcoded **f**(**o**) has a resolution of 400x300. The applied paint strokes containing the absolute cartesian coordinates of the paint lines would therefore be inaccurately applied to the wrong location in the original **o**. In Figure 5 we see that this is indeed the case: the circular paint strokes can be seen above and to the left of the butterfly.

In order to properly implement the reconciliation rules for this application, the programmer can use the evaluation table we showed earlier. The first line of Table 1 corresponds to this imaging problem. Here, the programmer takes into consideration the image resolution reduction transcoding and the paint stroke updates to derive that the **T** transformation must appropriately scale the strokes up. He then considers the evaluative condition using visual inspection to see that the paint strokes can be applied correctly.

Figure 6 shows this approach where the operation **g** is first transformed to **G**. Here, the transform function **T**, which as we have said is a heuristic related to **f** and **g**, is derived by considering that the original **f** function reduced the image from 800x600 to 400x300 and that the **g** operation of applying paint strokes was performed on the

400x300 image. An appropriate **G** should thus be applied on the original 800x600 photo. The **T** function therefore appropriately scales up each paint stroke's cartesian coordinate endpoints, initially appropriate for the 400x300 size of **f**(**o**), to instead match the original 800x600 size of **o**. Figure 6 therefore shows the paint strokes, after transformation, scaled up to be drawn around the butterfly. For completeness, Figure 7 shows the image after it has been transcoded to form the subsequent **f·G**(**o**) object. Here, the image has been transcoded again; as we have said, we use the same transcoding **f** to simplify the presentation. However, now clearly the transcoding function operates on the applied paint strokes as well as the image. Most importantly, the choices of **T**, **g**, and **f** satisfy the property **g·f**(**o**) = **f·G**(**o**) as much as possible and can be seen visually by comparing Figure 3 and Figure 7. (Note how the circling has now lost quality. The circle would not have lost as much quality had **T** been implemented more completely, namely by increasing the thickness of each stroke in addition to scaling its endpoints.) Furthermore, comparing Figure 7 and Figure 5 shows that the latter's naive implementation is clearly incorrect after transcoding.

iDraw also allows the user to annotate the image by entering text directly on the image. For brevity, we omit a screenshot of this capability, but we note this **g** operation can also be transformed into a **G** by scaling the applied text similarly as we did with the paint strokes. However, operations that add information like this textual annotation raise another issue. Suppose the user annotates the image with

something as seemingly innocuous as "The green leaves look nice" and then performs a transfer from a PC to a greyscale PDA. After the image is transcoded for the PDA by reducing the colour to shades of grey, the annotation will have lost its meaning entirely. The burden of managing such operations falls on the user.

5.5   Other examples

In the butterfly example, we derived the $\mathbf{T}$ transform by considering the fact that because the transcoding function $\mathbf{f}$ scaled an image down in size, the corresponding $\mathbf{T}$ must scale the user updates up to match. However, note how we did not address the problem of the colour depth reduction. In our implementation, we sidestepped the issue by using black paint strokes all the time. As mentioned earlier, finding a "reverse" function for a many-to-few colour depth reduction is impossible unless additional information is available (e.g., we could define that re-colourising a colour depth reduction always results in one colour, as we had done with black). This approach is shown in the third line of Table 1. Similar difficulties may be encountered in other applications using different $\mathbf{f}$ and $\mathbf{g}$. Some transcodings, such as scaling, have a natural $\mathbf{T}$, while others, such as colour changes, do not. For the purposes of a graphics program, basic graphics theory tells us that in some cases $\mathbf{T}$ is trivially found because some operations satisfy the property $\mathbf{f} \cdot \mathbf{g(o)} = \mathbf{g} \cdot \mathbf{f(o)}$; that is, some operations are mutually independent and commutative. In these cases $\mathbf{T}$ is thus found easily if (1) $\mathbf{f}$ and $\mathbf{g}$ are both geometric translation, rotation, or scaling operations or (2) if $\mathbf{f}$ is a uniform scaling function and $\mathbf{g}$ is a geometric translation. As a negative example, if $\mathbf{f}$ were a resolution reduction and $\mathbf{g}$ were an image-blurring filter, then these operations would not be mutually independent; furthermore, it is not clear in this case that for this $\mathbf{f}$ a $\mathbf{T}$ could be found that translates $\mathbf{g}$ into $\mathbf{G}$ and satisfies $\mathbf{g} \cdot \mathbf{f(o)} = \mathbf{f} \cdot \mathbf{G(o)}$.

Table 1 also shows possibilities with other programs. Consider a user application that can accept Adobe PDF files that can be transcoded by a PDF-to-text filter to suitably fit the client. Upon receiving text, the user annotates it by underlining some words. An appropriate $\mathbf{T}$ function would transform these $\mathbf{g}$ operations into a $\mathbf{G}$ that can act upon the original PDF format by changing underlining on text at the client into underlining on the PDF data at the proxy. However, we must still check if $\mathbf{g} \cdot \mathbf{f(o)} = \mathbf{f} \cdot \mathbf{G(o)}$ is true. If the transcoder filters underlined PDF into underlined text, then the system satisfies our semantics. Similarly, consider a speech-to-text transcoder. If a $\mathbf{T}$ transform changes underlined text to raised-volume speech and if the transcoder filters raised-volume speech to underlined text, then again, the system complies. Our approach in these and other application domains is an area of future work.

6.   MIDDLEWARE IMPLEMENTATION

In this section we provide a detailed look at MoxieProxy[1], a complete middleware architecture and software toolkit to support the reconciliation rules and design methodology we developed in the previous section. Broadly speaking, we utilise the three-tier architecture shown in Figure 4 where a middleware proxy is placed between application servers and heterogeneous clients. Our middleware tier runs a set of generalised services (including transcoding, caching, and

service discovery) as well as application-specific code (for handling program logic and protocols), all written in Java. Furthermore, the middleware proxy can be run on top of a workstation cluster to improve scalability. At the client side, an application must include a Java object that provides an interface to the middleware services running on the proxy.

6.1   Client-side components

In Figure 8 we show a block diagram of the client-side software architecture for an application modified to interoperate within our architecture. The diagram shows that the program maintains content that is defined in an application-specific manner. Furthermore, we assume that the source code of the applications can be modified to include a Java object we call the Middleware-Aware Remote Code, or MARC; this assumption follows from our suggestion that the programmer and the service provider collaborate. By encapulating our code into a single Java object, the application's original functionality is maintained without interference from either the imported MARC object or the proxy.

MARC chassis. The MARC object operates within a framework chassis and initially comprises only two functionalities. First, a service discovery functionality can find the nearest proxy using a discovery protocol. Second, the proxy responds by sending an application-specific module that contains the application logic allowing the program to utilise content transfer.

The lazy, on-demand downloading of the module to the client provides two important features: it allows the MARC to maintain the smallest possible footprint until full functionality is needed, and it facilitates software maintenance by allowing updates to be distributed at the proxies instead of the clients. Furthermore, because the client-side module is downloaded as a Java object at runtime, it can be cached and flushed appropriately by the Java virtual machine as needed, thereby granting a flexibility that reduces memory usage for the program. This dynamic downloading is an example of the remote object factory design pattern.

Our MARC operates smoothly with applications thanks to its clean object-oriented design. Utilising the MARC with Java applications is straight-forward; we have also had prior success incorporating Java MARC objects into C++ code by using Sun's Java Native Interface Java-C++ glue.

Application-specific modules (ASM) and the ASM loader. The application-specific modules are the key software components that allow us to flexibly implement our reconcilitation rules. An ASM is the interface between the original application and our middleware proxy. In the absence of the ASM, an application communicates with its application server using an agreed-upon protocol (e.g. HTTP for web, SMTP for mail, or ODBC for a DBMS). In our environment, a client-side ASM communicates with a proxy-side ASM, which in turn communicates with the application server using the application protocol.

At the client, the ASM library code provides an API for the application to receive the $\mathbf{f(o)}$ object from the proxy and to send the client update $\mathbf{g}$ back to the proxy. Each application-specific module (as its name implies) must be designed to the particulars of a given application. Communication between the client and the proxy is conducted using Java's RPC mechanism, Remote Method Invocation (RMI). We chose RMI as our standard connectivity API instead of socket streams because we wanted to keep the communica-

---

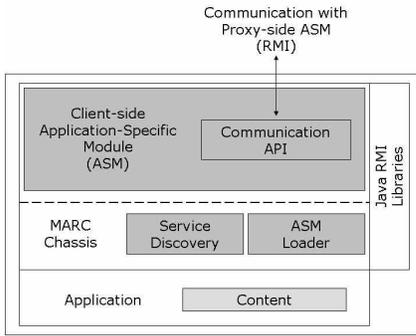[1]moxie (noun): 1. energy, pep; 2. courage, determination

Figure 8: Client-side architecture. A user application contains a MARC Java object that can load client-side application-specific modules (ASMs) on-demand.
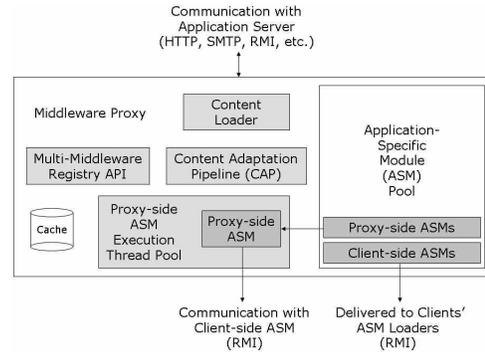

Figure 9: Proxy-side architecture. A middleware proxy contains a number of components that run between an application server and a client.

tion at the granularity of objects, not bytes. That is, the ASM-enabled application requests and hands off whole objects like images and documents instead of streams of bytes, thereby obviating the need for content parsing to assemble and disassemble packet streams. Furthermore, although objects can also be sent via Java sockets, RMI simplifies the programming interface considerably. This approach is similar to previous architectures that used object-based mobility, such as Emerald [18] and Rover [17].

The loader allows an ASM to be downloaded from the middleware proxy and installed dynamically. We implemented this dynamic software downloading by also using Java RMI to get class files and a custom class loader to install the class object into the running virtual machine.

Service discovery. In self-organised mobile networks, automatic service discovery is critical. We implemented our service discovery component using Jini to allow applications to find and interoperate with middleware proxy services on the network. Jini provides a Java API to look up services based on registries and service leases.

6.2 Proxy-side components

In Figure 9 we show the software architecture within MoxieProxy's middleware to support the application. This design is essentially a mirror image of the architecture used at the client's side; here, proxy-side ASM code is loaded dynamically and run to complement the activity at the client-side ASM. Content is also cached appropriately.

Application-specific module pool. The proxy maintains a pool of client-side and proxy-side ASMs for each supported program. For example, in our implementation, the modules for the iDraw and iShare applications are managed as separate Java `.class` files. Client-side ASM code is delivered on-demand to the client as we discussed earlier. Proxy-side ASM code is similarly loaded on-demand into an execution thread pool, as we discuss next. Each proxy-side ASM performs the functionality needed at the proxy for the reconciliation techniques we developed; these ASMs are responsible for calling the content loader to download content from the application server, calling our transcoder to adapt the data, transporting content to the client, receiving **g** operations from the client, transforming **g** to **G**, and applying these updates. Although the final step of applying updates

could potentially be done at the second client (by shipping the transformed operation, in XML representation, to the client), the application should be done at the proxy because in general, the proxy has more computational resources than the clients and may be better suited to perform the update.

ASM execution thread pool. A proxy-side ASM is loaded into one of several pre-allocated execution threads. Upon initialisation of the proxy, the system starts a variable number of threads that can be adjusted by the service provider. With each thread in the pool handling a client request, the performance of the proxy scales well since the ASM blocks on network I/O when interacting with either the application server or the client. One problem we found is that our transcoding mechanism is computation-bound and negates much potential concurrency on a uniprocessor.

Content loader. This component loads data from the application server. Although it we could have incorporated this functionality into an ASM, we decided to make this component separate in order to share common loading protocols among different applications. For example, the iDraw program can upload and download images with a web server or with our custom RMI-based server. Accordingly, there is a separate loader for HTTP and RMI; the RMI loader is also used by the iShare program. In the past we have also implemented loaders to handle SMTP for a mail application as well as RTP for a streaming multimedia program.

We must also consider how to treat these objects if the user commits changes back to the application server. For transcoding-independent **g** operations, the resulting object **G(o)** has the same data type as the original object **o** since it was simply modified with a legal operation. This fact can be manifest by the default action taken by the application and proxy when the user wishes to commit changes: the system defaults to overwriting the original object **o**. On the other hand, for transcoding-dependent **g** operations, the object **g·f(o)** should not be considered of the same type as **o** due to the use of **f**. In fact, as we have said, a transcoding operation can transform an object from one data type to another, such as a speech-to-text transcoder. When the user commits his changes, the system defaults to saving to a different file.

Content Adaptation Pipeline. Our transcoding mechanism, the Content Adaptation Pipeline (CAP), divides the transcoding process into separate extensible stages [29]. Ini-

tially, when the client registers itself to our system, the client profile (encoded in XML) is sent to and stored at the proxy to provide the necessary parameters to perform transcoding. The profile includes parameters such as the device's display, memory, runtime libraries, network interface, and CPU. After this registration, the client application can then request a data object from the application server via the proxy. To identify the data object, we pass it through a Data Characterisation Function to determine its data type and characteristics. These resulting characteristics (in XML) and the client profiles (and optionally the current network conditions) are passed as inputs into the Adaptation Command Generator to produce a set of commands (again in XML) specifying how the object should be adapted to best match the client's constraints. These commands are then parsed by the Content Adaptation Executor, which will call the appropriate routines within its pool of available adaptation library functions to perform the actual transcoding on the previously cached object. Finally, the adapted object is delivered to the client.

In Figure 10 we quantify the overhead incurred by the CAP to transcode a JPEG image by reducing its resolution and colour depth. Here we executed the CAP on a Sun UltraSPARC workstation on a fast ethernet LAN. From the graph it is clear that the processing time for the CAP increases with the size of the original image, a typical trait in transcoding mechanisms. This figure shows that even for the relatively large size of 300 kbytes, the completion time of the CAP is acceptable, under 1600 milliseconds.

Multi-middleware registry for clustered operation. Although the focus of our work is not scalable performance, for completeness and robustness we can have the proxy optionally execute across a local-area cluster of workstations [1] [12]. Our design is similar to that used in modern load-balanced Web farm clusters. Each client communicates with a front-end proxy that maintains a registry of participating back-end Sun Solaris workstations in the cluster. The front-end proxy chooses a participating workstation in a round-robin fashion from the registry and returns the IP address of the chosen workstation to the client, which then interacts directly. The workstations, each running an instantiation of our middleware proxy software, autonomously decide if they have sufficient computational resources to provide service to clients. We use as a metric the CPU load provided by the UNIX library call `getloadavg(3C)`, which returns the number of processes in the system run queue. If the workstation's load is over a chosen threshold, it autonomously decides to deregister itself from the front-end proxy, which removes the workstation from the registry of available servers. Such an approach scales well since the decisions are made locally at each workstation without requiring the centralised front-end proxy to maintain excessive state or poll each server.

To show how our clustered system provides scalability, we show results in Figure 11 that reflect the execution of the CAP distributed over three proxies within a heterogeneous cluster of Sun UltraSPARC workstations. Our front-end middleware proxy was a Sun Blade 100 workstation running at 501 Mhz with 1 GB of RAM. We structured the CAP to perform JPEG transcoding as before. However, in this case we placed the transcoder under stress by progressively increasing the service load demanded by clients to test our load-balancing scheme across our back-end cluster.
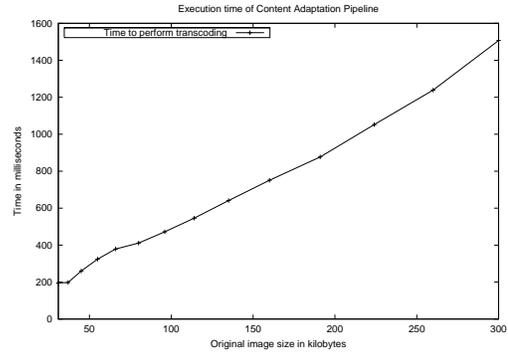

Figure 10: Average execution time of the CAP to transcode a images as a function of increasing image size. The target profile was for an iPAQ PDA with 256 colours and a 300x200 screen. The original images were 24-bit colour of varying resolution.
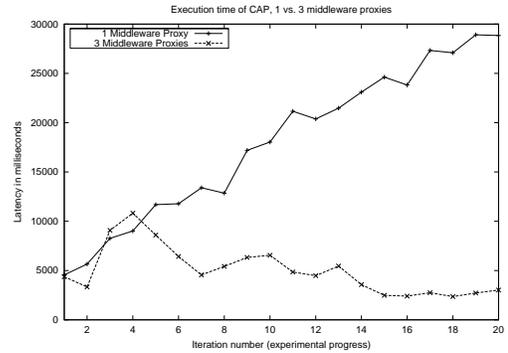

Figure 11: CAP latency on multiple middleware proxies.

The figure shows the latency to perform the transcoding function as experienced by a given client. In the experiment we launched a new client (requesting a 500 kbyte JPEG) at random intervals based on a Poisson distribution with a mean of 15 seconds. Upon launching, each client requests data from the application server via the proxy and continues to make such requests at random Poisson intervals with a mean of 30 seconds. We localised our measurements on one client, so therefore, each tick on the x-axis represents one request iteration for this client (after a random interval).

The data plot for the 1-proxy case shows that the latency seen by the client grows without bound as more clients (and their requests) are introduced. This growth is due to the fact that the proxy must perform the computationally-bound JPEG transcoding. For the 3-proxy case, we implemented a load distribution policy of having each proxy monitor its current CPU load. When its load passes a chosen threshold (specified by the proxy administrator), the proxy refuses further service. In the case of the client in this experiment, it is redirected to another proxy. With this easily implemented scheme, the client's completion time grows and then falls off, showing that the self-regulating load-balancing scheme provides reasonable scalability.

7. CASE STUDIES

The **iDraw** application, as we have seen throughout the paper, is an image-editing/paint program that allows the user to perform a number of different paint operations, such as multi-colour paint strokes and text annotations, as well as image filters, such as sharpening, blurring, colour inversion, and edge detection. The program was written in Java as a stand-alone Swing/JFC program using JDK 1.3. In our work, we used iDraw to demonstrate content migration by having its content sent from one device to another via our middleware proxy.

In addition to handling images stored on local disk, iDraw can also download and upload images over the Internet. Images can be retrieved from a Web server via HTTP (by supplying a URL) and then sent back to servers running a CGI script we have written. Furthermore, the program can interact with our custom RMI-based image server that publishes the requisite Java interface for down/uploading images; in this case, we use a special serialisable Image class because the default Java Image class cannot be serialised.

The **iShare** content-sharing whiteboard application has essentially the same GUI drawing functionality as iDraw, except it does not have the image filter operations. Like iDraw, it can down/upload images over the Internet. We note that although much work has been done at the network layer to provide reliable, consistent multicast whiteboard operations (e.g. [10]), we used a very simple scheme because we are focused on high-level semantics. Consequently, iShare's whiteboard capability is implemented via an application-layer multicast facilitated by a proxy that acts as a multicast point. Even in the absence of our middleware architecture, iShare operates with its multicast proxy using the RMI callback mechanism to allow operations at one client to be conveyed to other whiteboard users who have registered themselves as participants at the multicast proxy. Updates from multiple clients are managed serially and atomically via Java's built-in "synchronized" keyword on the remote methods, thereby enforcing monitor behaviour.

In our experiments, our single middleware proxy was a 501 Mhz Sun Blade 100 workstation with 1 GB of RAM. Our application/WWW server was a Sun UltraSPARC-5 workstation running at 270 Mhz with 128 MB of RAM. Our test client was a Windows 2000 Pentium 4 running at 1.5 Ghz with 256 MB of RAM. All the machines were connected via fast ethernet on a LAN.

7.1 Client-side middleware support

Both iDraw and iShare are easily modified to incorporate a client-side MARC object. The MARC chassis is instantiated and treated just like any other Java object. A service-discovery method can then be called to find an available middleware proxy, and an appropriate application-specific module can then be downloaded into the chassis. The ASM is named using a unique identifier associated with the application. Within the ASM, a variety of operations related to the semantic reconciliation rules are performed as we enumerated earlier, including user registration, content downloading, and **g** uploading. The operations are accessed as ASM object method calls that must be provided by the application programmer and/or service provider.

As we noted earlier, uploading just the **g** operation instead of the larger **g·f(o)** object can result in substantial savings. In Table 2 we show the data sizes for a sample 1280x960

image, set of brush strokes, and text annotations. Since the brush strokes and annotations can be encoded as XML text, they can potentially be significantly smaller. With low-bandwidth connectivity, this is an optimisation with tangible benefits.

Data	Size
g · f(o) image	120 kbytes
g brush strokes	40 kbytes
g text annotations	2 kbytes

Table 2: Sizes of sample data sent from the client to the proxy upon content transfer. Note how the separation of the g operations from the underlying image substrate has the potential to greatly reduce the data to be uploaded. It is also worth noting the brush strokes were encoded in XML but stored in a Java Vector, incurring some overhead.

7.2 Proxy-side middleware support

7.2.1 Approach

The proxy-side support for iDraw (to perform content migration) and iShare (to perform content sharing) can be modularised according to the internal middleware proxy organisation. The proxy-side ASM delegates responsibility to the content loader to handle HTTP with a Web server or RMI with our special image server. Received content is then passed to the CAP for transcoding. Execution can optionally be load-balanced across a workstation cluster using our multi-middleware proxy design. The above actions are common to both applications.

For iDraw, the proxy-side ASM performs functionality specific to content migration. First, the application logic required to apply transcoding-independent **g** imaging operations is duplicated within the proxy-side ASM, but this imaging engine cannot be used alone. Instead, following the design methodology previously described, we identified the transcodings available for the image types that iDraw can handle (such as colour depth and resolution reduction). A subsequent **T** functional was then derived for **f** and **g** pairs, and object methods were coded to create the resulting transcoding-independent $\mathbf{G} = \mathbf{T}(\mathbf{f}, \mathbf{g})$. For this application, the implementation of **T** included the scale-up transformation we demonstrated earlier. The coding involved a few dozen lines of Java that calculated the correct scaling factor and looped over all the paint strokes to scale each one correctly, resulting in the transformed **G**. Thus, it is this final **G** that is used within the proxy-side ASM for iDraw. The result is that this ASM properly handles the reconciliation steps during the content migration.

For iShare, the proxy-side ASM functionality was created using similar steps to create the required **G** operations needed to reconcile updates for all users in the whiteboard session. The ASM also contains functionality specific to iShare's multicast requirements. The ASM implements a simplified multicast protocol and serves as a multicast point among all the participants. In particular, it maintains a soft-state list of users in the session and conveys updates between each client. **g** updates from a client are sent to the proxy-side ASM via RMI, and resulting **G(o)** objects are sent to other participants via RMI callback. Here, the reconciliation rules play a vital role. When a user updates the
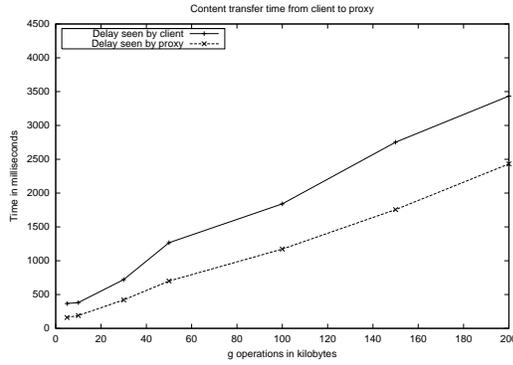
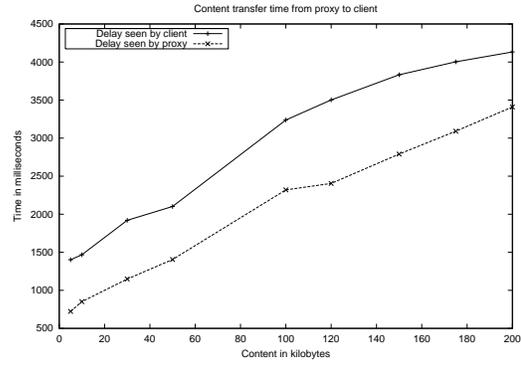**Figure 12: Average time to upload a g operation from a client to the proxy.**

Figure 13: Average time to download an f·G(o) object from a proxy to a client.

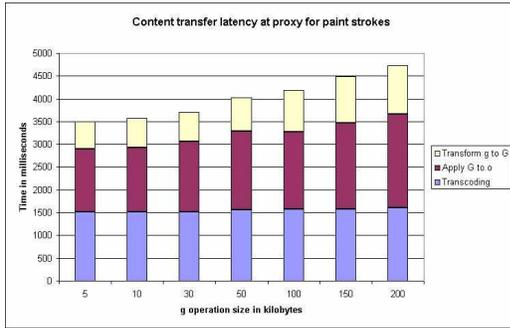**Figure 14: Average time to transform a g paint stroke to G, apply it to an o, and transcode it to f·G(o). The original JPEG was a 300 kbyte, 24-bit colour image; the transcoder was set to reduce the colour depth and image size to fit an iPAQ PDA with 255 colours and 300x200 screen.**
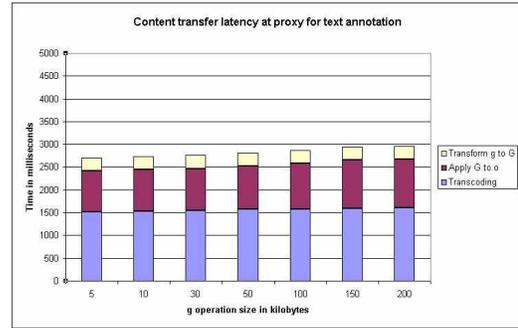

Figure 15: Average time to transform a g text annotation to G, apply it to an o, and transcode it to f·G(o). Transcoding parameters were the same.

whiteboard content, the ability to apply **G** directly to **o** results in two substantial advantages. First, as we have seen, it allows the other participants to receive the highest-fidelity content available without the transcoding required for the first client. Second, with multiple users, the creation of **G(o)** obviates the need to maintain separate transcoded versions appropriate for each client; that is, the ASM only maintains (**o**) rather than a collection of $f_1(o)$, $f_2(o)$, $f_3(o)$, etc. This simplification grants a savings not only in disk space but also file management; with a large number of participants requiring different transcoding support, this approach is a substantial advantage. The updated whiteboard content is then transcoded appropriately for the other clients and delivered. Concurrent updates in iShare from multiple users are applied serially at the proxy. Each client uploads its g_i update to the proxy, which in turn applies the appropriate G_i updates in a serial manner. The resulting object is then delivered to other users.

7.2.2 Experimental results

In order to quantify the performance of the proxy to perform the operations needed to execute the reconciliation rules, we ran a series of experiments that were common to both iDraw and iShare. First, in Figure 12 we show the average baseline overhead incurred by the Java RMI communication library. In this graph we plot the time to complete a transfer of a **g** operation from the client to the proxy as a function of increasing **g** size, as seen by both the client and proxy. Note that since RMI follows remote procedure call behaviour, the completion time at the client is subsumed by the completion time at the proxy. The latencies include the time for communication and writing to disk. (The reconciliation operations are not included here.) In this experiment we used paint strokes as the **g** operations being uploaded; the cumulative strokes are stored in a Java Vector, serialised (marshalled) into a transferable form by the Java RMI communication library (thereby incurring some payload overhead), and delivered to the proxy. As expected, transferring larger **g** operations results in a longer average communication delay.

In Figure 13 we measure the time to move content from the proxy to a second client. Here, the data is the **f·G(o)** object after applying reconciliation rules. It can again be seen that the download time increases with the size of the data being sent. We note that the time to perform the downloading is up to 1.4 times slower for a given object size than the corresponding upload of similar size shown in the previous figure. We believe that the time to do an upload (which includes a write to disk) completes faster due to the Java I/O write buffers at the proxy.

In Figure 14 we show the average time incurred to transform uploaded **g** paint strokes into **G**, apply it to the original **o** JPEG image, and then run it through the CAP transcoder to attain **f·G(o)**. Specifically, we transformed each paint stroke received from the client into an appropriate transcoding-independent operation by scaling each stroke's Cartesian coordinates up following the approach we discussed earlier. The resulting **G** operation was applied onto an original JPEG image (of size approximately 300 kbytes) and then transcoded. The graph shows that the average time to complete these procedures increases with the size of the **g** operations, an intuitive trend due to the computation-bound nature of these operations. Even at 200 kbytes of layered content, the latency of under 5 seconds is acceptable. With a more optimised implementation (particularly within the transcoder), this latency will decrease. Note that the time to perform the transcoding does not grow as much with increasing **g**; the **G** operations are merged into the original JPEG, resulting in an updated image still approximately the same size as before, thereby incurring similar CAP transcoding time. This similarity may not always be the case, as it is possible that the **g** operation could add significantly large amounts of content such that the resulting **G(o)** and **o** objects may differ drastically in size.

In Figure 15 we show the results of a similar experiment, this time using text annotations for **g**. Here, the text annotations are collectively controlled as an aggregate data string and are scaled up appropriately by changing variables in the text's properties. From the graph, it is clear that this transformation incurs less latency than scaling up paint strokes, which must be done on an individual basis for each stroke. Similarly, the time to apply the text annotations to the image is also smaller because the text is applied collectively, whereas paint strokes must be done individually. We expect that other applications will have different operational latencies as well, depending on the nature of the applications.

8. CONCLUSION

During a content transfer, much semantic information in the application's content can be lost due to the transcoding of downloaded data to fit the limitations of client devices and bandwidth. When a user performs update operations on such data, the user is explicitly acting on the transcoded data whereas implicitly what he really wants is to act on the original data. Upon transfer to another device, the session is transcoded again, but unfortunately the user is presented with data limited by both the first and second transcodings. If the first transcoding produces more loss than the second, then much semantic value will be lost, and the content may quickly degenerate to its lossiest form as it is passed among other users and devices.

Our paper suggested a way around this problem. In the applications we are studying, user operations may be represented by layerings that can act upon substrates. This capability allows the operations to be separated from its operand and transformed into another operation that can be applied directly to the original object instead of to the transcoded object. We say that operations such as these are transcoding-independent. The end result is that the disjoint nature of transcoding-independent operations allows the first transcoding to be omitted upon transfer, thereby allowing more semantic information to be retained.

In future work the results in this paper can provide the conceptual framework for evaluating the effectiveness of content transfer for other applications and transcodings. Additionally, we look to resolve several open issues. First, we will analyse the performance-storage tradeoff of our reconciliation mechanism. Second, we will look into ways of handling data objects that are too big to be downloaded in their entirety before being manipulated. Although user updates might only be able to operate on entire objects, in some cases it may be possible to stream objects by first decomposing them them into smaller sub-objects and then sending them to the user in a piece-wise manner, an approach similar to that in [8]. Third, we will investigate means for allowing users to specify how decoupled operations should be transformed. In much the same way that a transcoding mechanism should allow the user to state preferences for transcoding data, a reconciliation mechanism should allow users to state parameters for transforming operations. A clear difficulty is then to find ways to capture this open-ended intent-based information; an XML-based representation would be one direction. Fourth, we will carry out a human interactivity study of the perceived improvements from the different reconciliation approaches mentioned. Finally, we will look to address end-to-end security with a PKI-based approach.

9. ACKNOWLEDGEMENTS

10. REFERENCES

[1] T. ANDERSON, D. CULLER, AND D. PATTERSON. "A Case for NOW (Networks of Workstations)," *IEEE Micro*, 15(1), pp. 54-64, February 1995.

[2] R. BAGRODIA, S. BHATTACHARYYA, F. CHENG, S. GERDING, G. GLAZER, R. GUY, Z. JI, J. LIN, T. PHAN, E. SKOW, M. VARSHNEY, AND G. ZORPAS. "iMASH: Interactive Mobile Application Session Handoff," In *Proceedings of* MOBISYS, May 2003.

[3] G. BANAVAR, J. BECK, E. GLUZBERG, J. MUNSON, J. SUSSMAN, AND D. ZUKOWSKI. "Challenges: An Application Model for Pervasive Computing," In *Proceedings of* MOBICOM, August 2000.

[4] F. BANCILHON AND N. SPYRATOS. "Update Semantics of Relational Views," *ACM Transactions on Database Systems*, 6(4), pp. 557-575, December 1981.

[5] F. BENTAYEB AND D. LAURENT. "View Updates Translations in Relational Databases," In *Proceedings of the International Conference on Database and Expert Systems Applications*, August 1998.

[6] E. BREWER, R. KATZ, E. AMIR, H. BALAKRISHNAN, Y. CHAWATHE, A. FOX, S. GRIBBLE, T. HODES, G. NGUYEN, V. PADMANABHAN, M. STEMM, S. SESHAN, AND T. HENDERSON. "A Network Architecture for Heterogeneous Mobile Computing," *IEEE Personal Communications*, 5(5), pp. 8-24, October 1998.

[7] Y. CHAWATHE, S. FINK, S. MCCANNE, AND E. BREWER. "A Proxy Architecture for Reliable Multi-

cast," In *Proceedings of the ACM International Conference on Multimedia*, September 1998.

[8] E. DE LARA, R. KUMAR, D. WALLACH, AND W. ZWAENEPOEL. "Collaboration and Multimeda Authoring on Mobile Devices," In *Proceedings of* MOBISYS, May 2003.

[9] C. ELLIS AND S. GIBBS. "Concurrency Control in Groupware Systems," In *Proceedings of* SIGMOD, May 1989.

[10] S. FLOYD, V. JACOBSEN, S. MCCANNE, C. LIU, AND L. ZHANG. "A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing," In *Proceedings of* SIGCOMM, August 1995.

[11] A. FOX, S. GRIBBLE, Y. CHAWATHE, AND E. BREWER. "Adapting to Network and Client Variability via On-Demand Dynamic Distillation," In *Proceedings of* AS-PLOS, October 1996.

[12] A. FOX, S. GRIBBLE, Y. CHAWATHE, E. BREWER, AND P. GAUTHIER. "Cluster-Based Scalable Network Services," In *Proceedings of* SOSP, October 1997.

[13] R. GRIMM, J. DAVIS, B. HENDRICKSON, E. LEMAR, A. MACBETH, S. SWANSON, T. ANDERSON, B. BERSHAD, G. BORRIELLO, S. GRIBBLE, AND D. WETHERALL. "Systems Directions for Pervasive Computing," In *Proceedings of* HOTOS, May 2001.

[14] R. HAN, P. BHAGWAT, R. LAMAIRE, T. MUMMERT, V. PERRET, AND J. RUBAS. "Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing," *IEEE Personal Communications*, 5(6), pp. 8-17, December 1998.

[15] V. JACOBSEN. "A Portable, Public Domain Network 'Whiteboard'," *Xerox PARC, viewgraphs*, 1992.

[16] B. JOHANSON, S. PONNEKANTI, C. SENGUPTA, AND A. FOX. "Multibrowsing: Moving Web Content Across Multiple Displays," In *Proceedings of* UBICOMP, September 2001.

[17] A. JOSEPH, J. TAUBER, AND M. KAASHOEK. "Mobile Computing with the Rover Toolkit," *IEEE Transactions on Computers*, 46(3), pp. 337-352, March 1997.

[18] E. JUL, H. LEVY, N. HUTCHINSON, AND A. BLACK. "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems*, 6(1), pp. 109-133, February 1988.

[19] A. KELLER. "The Role of Semantics in Translating View Updates," *IEEE Computer*, January 1986.

[20] G. KUENNING AND G. POPEK. "Automated Hoarding for Mobile Computers," In *Proceedings of* SOSP, October 1997.

[21] P. KUMAR AND M. SATYANARAYANAN. "Flexible and Safe Resolution of File Conflicts," In *Proceedings of the USENIX Winter Technical Conf.*, January 1995.

[22] Y-W. LEE, K-S. LEUNG, AND M. SATYANARAYANAN. "Operationed-based Update Propagation in a Mobile File System," In *Proceedings of USENIX*, June 1999.

[23] M. LITZKOW, M. LIVNY, AND M. MUTKA. "Condor - A Hunter of Idle Workstations," In *Proceedings of* ICDCS, June 1988.

[24] W. LUM AND F. LAU. "On Balancing Between Transcoding Overhead and Spatial Consumption in Content Adaptation," In *Proceedings of* MOBICOM, Sept. 2002.

[25] S. MCCANNE. "A Distributed Whiteboard for Network Conferencing," *UC Berkeley class report*, 1992.

[26] D. MILOJICIC, F. DOUGLIS, Y. PAINDAVEINE, R. WHEELER, AND S. ZHOU. "Process Migration," *ACM Computing Surveys*, 32(3), pp. 241-299, September 2000.

[27] B. NOBLE, M. SATYANARAYANAN, D. NARAYANAN, J. TILTON, J. FLINN, AND K. WALKER. "Agile Application-Aware Adaptation for Mobility," In *Proceedings of* SOSP, Oct. 1997.

[28] P. PARNES, D. SCHEFSTROM, AND K. SYNNES. "WebDesk – Collaboration Support in MATES," In *Proceedings of the European Workshop on Global Engineering Network*, February 1996.

[29] T. PHAN, G. ZORPAS, AND R. BAGRODIA. "An Extensible and Scalable Content Adaptation Pipeline Architecture to Support Heterogeneous Clients," In *Proceedings of* ICDCS, July 2002.

[30] H. SHU. "Using Constraint Satisfaction for View Update Translation," In *Proceedings of the European Conference on Artificial Intelligence*, August 1998.

[31] K. TAKASHIO, G. SOEDA, AND H. TOKUDA. "A Mobile Agent Framework for Follow-Me Applications in Ubiquitous Computing Environment," In *Proceedings of the International Workshop on Smart Applications and Wearable Computing*, April 2001.