# Handling Client Mobility and Intermittent Connectivity in Mobile Web Accesses

**Purushottam Kulkarni**, **Prashant Shenoy** and **Krithi Ramamritham***
Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003
{purukulk,shenoy,krithi}@cs.umass.edu

**Abstract**

*Wireless devices are being increasingly used to access data on the web. Since intermittent connectivity and client mobility are inherent to such environments, in this paper, we examine the impact of these factors on coherent dissemination of dynamic web data to wireless devices. We introduce the notion of eventual-delta consistency, and in the context of push and pull-based dissemination propose (i) proxy-based buffering techniques to mask the effects of client disconnections and (ii) application-level handoff algorithms to handle client mobility. Our experimental evaluation demonstrates that push is better suited to handle client disconnections due to its lower message overhead and better fidelity, while pull is better suited to handle client mobility due to its lower handoff overheads.*

## 1 Introduction

### 1.1 Motivation

Recent advances in computing and networking technologies have led to a proliferation of mobile devices such as mobile phones, PDAs and laptops with wireless networking capabilities. Like traditional wired hosts, wireless mobile devices are frequently used to access data on the web. Such mobile devices, however, differ significantly from their wired counterparts with respect to their capabilities and characteristics—(i) mobile clients can move from one location to another, while the location is fixed for wired hosts and (ii) disconnections and reconnections are frequent in mobile environments, while network failures are an exception in the wired domain. A concurrent trend is the increasing use of dynamic web data in today's web. Unlike static web pages that change infrequently, dynamic web data is time-varying and changes frequently (once every few seconds or minutes). Examples of such dynamic (i.e., time-varying) data include financial information such as stock prices, up-to-the-minute sports scores, weather and real-time traffic information. A key challenge in the design of applications that access dynamic data (e.g., stock tickers, weather applets) is to ensure that displayed (or cached) values are *temporally coherent* with the data source.

In this paper, we address the challenges arising from the intersection of these two trends, namely, the access of dynamic web data using networked mobile devices. Specifically, we examine the impact of intermittent connectivity and client mobility—two key characteristics of networked mobile devices—on maintaining coherency of dynamic data.

*Also with the Indian Institute of Technology Bombay, India

## 1.2 Problem Formulation and Research Contributions

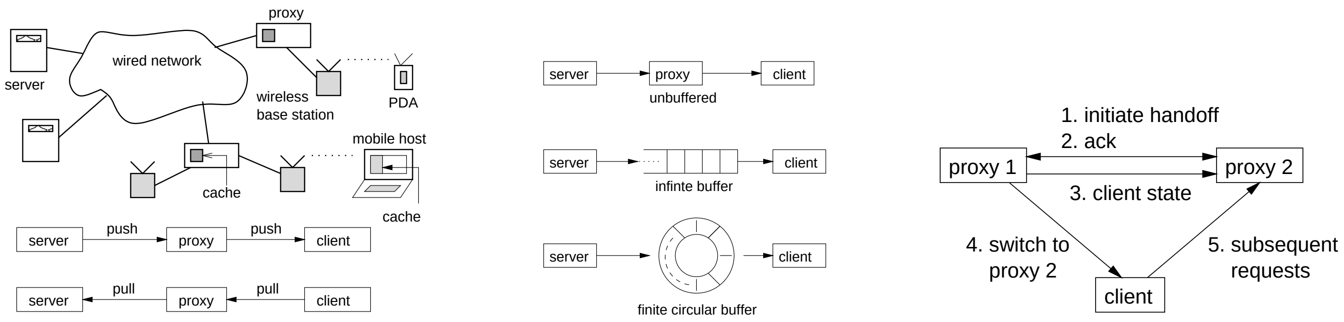
Consider a mobile client that accesses dynamic data items over a wireless network (see Figure 1(a)). Like in many wired environments, we assume that client web requests are sent to a proxy. The proxy either services the request from its cache (in the event of a cache hit) or fetches the requested object from the server (in the event of a cache miss). In addition to the proxy cache, the mobile client is also assumed to maintain a local cache (e.g., a browser cache) for performance reasons. Further, the proxy and the server are assumed to communicate over a wired network, while the proxy and the client communicate over a wireless network via a base station; the latter network can be either be a wireless LAN such as 802.11b or a wireless WAN such as a GSM data network. Each base-station in the wireless network is assumed to be associated with a proxy; different base-stations may be serviced by different proxies.

Intermittent connectivity and client mobility are common in such wireless environments; we discuss each issue in turn. We assume an environment where mobile clients can get disconnected at any time—disconnections occur primarily due to poor network coverage at a particular location. Disconnections can also occur due to the mobile device powering down to doze mode to save energy; such energy considerations are, however, beyond the scope of this paper. Since the proxy and the server are unreachable during a disconnection, dynamic data items cached at the client can not be refreshed and may become stale. Consequently, to prevent a violation of coherency guarantees, client disconnections should be detected in a timely fashion so that preventive measures can be initiated (e.g., by marking cached items as possibly stale). Further, the coherency of cached data items need to be restored upon a reconnection by resynchronizing the cache state with the proxy. Observe that, since the proxy and the server are on a wired network, the proxy can continue to receive updates from the server during a disconnection (but can not propagate these updates to the client). Hence, data cached at a proxy continues to be coherent even during a disconnection—an observation that can be exploited for efficient resynchronization of the client cache upon reconnection. Thus, coherency maintenance in the presence of intermittent connectivity requires (i) techniques for timely detection of client disconnections and reconnections, and (ii) techniques for efficient resynchronization of cache state at the client upon reconnection.

A second consideration in wireless environment is client mobility. Since a mobile client can move locations, the proxy may decide to hand over responsibility for servicing the client to another proxy in the client's vicinity. Such a *handoff* is desirable since a nearby proxy can reduce network overheads and provide better latency to client requests; the handoff involves a transfer of client-specific cache state from the initiator to the recipient proxy. Note that, handoff of client-specific state between proxies is different from handoff of network-level (connection) state that occurs between base stations—whereas the latter is implemented by the wireless network at the data-link or network layers, the former needs to be implemented by the proxies at the application layer. The design of such application-level handoff mechanisms involves two research issues. First, the proxy needs to decide *when* to initiate the handoff. This design decision has implications on the overheads imposed by handoffs and the latency seen by user requests. Second, we need to determine *how* the handoff occurs. For instance, the handoff process should be seamless to end-clients—the client should not miss any updates to cached data due to the handoff and all temporal coherency guarantees should be preserved. Further, the overhead of transferring client-specific cache state from one proxy to another should be minimized during a handoff. The design of application-level handoff mechanisms that address these issues is another goal of our work.

Thus, in this paper, we are broadly concerned with the implications of client mobility and intermittent connectivity on the dissemination of web dynamic data. We make three contributions in this paper. First, we propose coherency semantics appropriate for dynamic data access in mobile environments. We then consider two canonical techniques for dynamic data dissemination—push and pull—and show how to adapt these techniques to (i) reduce the overheads of coherency maintenance in the presence of client mobility and (ii) provide temporal coherency guarantees even in the presence of intermittent connectivity.

In the context of client mobility, we propose application-level handoff mechanisms for push- and pull-based data dissemination. We consider various variants of our handoff mechanisms that differ based on *when* state information




(a) System model (b) Unbuffered and buffer consistency (c) Handoff mechanism

**Figure 1**: System model and mechanisms for intermittent connectivity and client mobility.

is transferred during a handoff (referred to as *optimistic* and *pessimistic* handoffs) and *how much* state is transferred (referred to as *complete* and *partial* handoffs).

In the context of intermittent connectivity, we argue that the detection of disconnections and reconnections in a timely fashion is crucial to maintaining temporal coherency guarantees. To increase the resilience of push and pull to such network events, we propose algorithms for detecting disconnections and subsequent reconnections at a proxy and the end-client. We also propose mechanisms necessary at the proxy and the client to ensure that stale data is not used upon a disconnection and to ensure timely resynchronization of cached information.

We evaluate the effectiveness of our mechanisms using trace-driven simulations. Our results show that proxy-based buffering can help reduce the number of updates that are lost due to client disconnections. Further we find that a push-based approach provides a lower message overhead and better fidelity in the presence of disconnections, at the expense of maintaining client and proxy-specific state. In contrast, a pull-based approach incurs a lower handoff overhead and is better suited than push for handling client mobility.

The remainder of this paper is structured as follows. Section 2 defines the coherency semantics for mobile environments and proposes techniques for handling intermittent connectivity and client mobility. Variants of these techniques for push and pull-based dissemination are presented in Sections 3 and 4. Experimental results are presented in Section 5; related work is discussed in Section 6 and our conclusions are presented in Section 7.

## 2 Data Dissemination in Mobile Environments

### 2.1 Coherency Semantics for Mobile Environments

Due to the time-varying nature of dynamic data, cached versions of these data items need to be *temporally coherent* with the data source. To ensure this property, we define a new coherency semantics called *eventual-delta* consistency ($\Delta_e$) for mobile environments. $\Delta_e$ semantics provide stricter guarantees so long as a mobile client is connected and weaken the guarantee upon a disconnection. In particular, so long as a client remains connected, $\Delta_e$ semantics provide a bound on the amount by which a cached data item can be out-of-sync with the server. Upon a disconnection, this guarantee is weakened and the only guarantee provided to the client is that the coherency of cached items will be eventually restored (upon reconnection). Formally, $\Delta_e$ consistency is defined as follows

$$\Delta_e\text{consistency} \Rightarrow \begin{cases} |S_t - C_t| \leq \Delta & \text{if connected} \\ C_t \rightsquigarrow S_t & \text{if disconnected} \end{cases} \tag{1}$$

where $C_t$ and $S_t$ denote the state of a data item $d$ at the client and the server, respectively, at time $t$, $\Delta$ denotes the desired coherency bound, and $C_t \rightsquigarrow S_t$ indicates that $C_t$ will eventually catch up with $S_t$.

Depending on the application, $\Delta_e$ consistency can be enforced in the time domain, value domain or using version numbers. Time domain consistency requires that the client and the server are never out of date by more than $\Delta$ time units; value domain consistency requires that the value of a data item at the client and server is bound by $\Delta$; consistency using version numbers requires that the client and the server versions are no more than $\Delta$ apart. Coherency mechanisms that implement $\Delta_e$ consistency can be *buffered* or *unbuffered*. In the unbuffered scenario, each new update from the server overwrites the previously cached version—the proxy cache only maintains the version corresponding to the most recently received update. In the buffered scenario, the proxy buffers all recent updates from the server that are yet to be propagated to the client (see Figure 1(b)). Thus, all updates not yet seen by the client are buffered at the proxy and delivered to the client the next time it refreshes its cache. Since buffered $\Delta_e$ consistency can require a potentially infinite buffer, a practical approach is to bound the amount of buffer per data item by using a circular buffer of size $N$. In this case, the proxy buffers the $N$ most recent updates received between two successive client refreshes and overwrites prior updates (see Figure 1(b)). An appropriate choice of $N$ allows a smooth tradeoff between the unbuffered and buffered version of $\Delta_e$ consistency. If $N = 1$, the approach reverts to unbuffered consistency, while $N = \infty$ yields buffered consistency.

An attractive feature of $\Delta_e$ consistency is that the value of $\Delta$ can be chosen based on the coherency tolerances of the application—smaller values of $\Delta$ yield stricter guarantees. In the case where $\Delta = 0$, the approach provides strong consistency (which requires that a cached object be always up-to-date with the server). On the other hand, $\Delta > 0$ implies that a cached object may be out of date with the server by as much as $\Delta$. Moreover, if $\Delta > 0$, not every update to the data item at the server needs to be propagated to the proxy (or the client)—*only those updates that violate the bound $\Delta$ need to be propagated.* To illustrate, if a user is interested in changes of 25 cents or more to a stock price, smaller changes to the price need not be propagated by the server.

There are two possible techniques to implement $\Delta_e$ consistency—*push* and *pull*. In the push approach, cache coherency is maintained by pushing updates to dynamic data items from the server to the proxy (and from the proxy to the clients). In the pull approach, the onus is on the proxy (client) to periodically poll the server (proxy) and pull updates of interest—the more frequent the updates or the more stringent the value of $\Delta$, the more frequent the poll. The effectiveness of either approach in maintaining the desired coherency guarantees is measured using a metric referred to as *fidelity*. Fidelity is defined to be the normalized duration for which the desired coherency guarantees are maintained at a cache. The goal of a desirable coherency mechanism is to maintain high fidelity at low cost.

### 2.2 Handling Client Mobility using Handoffs

A typical wireless network consists of cells, each of which service a certain geographical region. As a mobile client moves from one location to another, the base-station in each cell hands over (link-layer) responsibilities for servicing the client to the base-station in the new cell. As explained earlier, each mobile client uses a proxy to request web content. In general, it is possible for a mobile client to use the same proxy to service its requests even when it moves locations (since it can always communicate with this proxy so long it remains connected). However, other proxies may be more advantageously located with respect to the new client location. In such a scenario, having the mobile client use a nearby proxy to service its requests may result in improved response times and lower network overhead. To enable such a transition, the proxy needs to hand over responsibility for servicing client requests to the new proxy. This involves handing over client-specific cache state maintained at the proxy so that the new proxy can continue to service the client in a seamless fashion.

Several issues arise in the design of such application-level handoff mechanisms. First, the proxy needs to be aware of the client's physical location so that it can determine whether a handoff is necessary. Several mechanisms already exist to track client location both in wireless LANS such as 802.11b [4, 7] as well as wireless WANs such as GSM data networks [13]. We assume that the proxy can employ these mechanisms to determine the client location and initiate a handoff when another proxy is better suited to service the client.

A second issue is how to ensure that the handoff is seamless and transparent to the end-client. A handoff is seamless so long as the client is minimally involved in the process. To ensure this property, the handoff mechanisms proposed in this paper employ the following basic structure. Assume a client $C$ that is currently using proxy $P_1$ to service its needs. Upon moving to a new location, assume that $P_1$ decides to hand over responsibilities for serving $C$ to another proxy $P_2$. The handoff involves the following steps (see Figure 1(c)): (1) $P_1$ contacts $P_2$ to initiate a handoff, (2) $P_2$ agrees to service $C$ if it has the resources to do so, (3) $P_1$ then transfers client-specific state to proxy $P_2$ (*what* state information is transferred and *when* depends on the specific handoff algorithm), (4) At this point, $P_2$ has the necessary information to service $C$; $P_1$ then notifies $C$ to send all further requests to proxy $P_2$. Note that, until this point, client $C$ is oblivious of the interactions between $P_1$ and $P_2$ and continues to interact normally with $P_1$. Once notified, the client simply switches to $P_2$ by sending subsequent requests to $P_2$ and begins receiving updates to cached items from $P_2$. Thus, the client is minimally involved in the handoff process and handoffs are largely transparent to the client.

A third issue in the design of handoff algorithms is when and how client-specific state information is transferred from the initiator proxy to the recipient proxy. Depending on the exact mechanisms, the following types of handoffs are possible:

- *Optimistic versus pessimistic:* In optimistic handoffs, the initiator proxy transmits both the meta-data for objects accessed by the client as well as the objects (data). The meta-data for an object consists of the object identifies (URL), size, expiration time, etc. Transferring both the meta-data and the data allows the recipient proxy to start with the same state as the initiator, and thereby reduce latencies for subsequent requests from the client. In pessimistic handoffs, only the meta-data is transferred from one proxy to another. In the event of a subsequent access, the new proxy must fetch the object from proxy $P_1$ (or from the server). Thus, objects are transferred lazily in pessimistic handoffs, while they are transferred in an eager fashion in optimistic handoffs. Consequently, optimistic handoffs incur a larger transfer overhead while providing potentially better response times; the opposite tradeoffs hold for pessimistic handoffs.

- *Complete versus partial:* Another dimension that differentiates handoff mechanisms is how much state information is transferred between proxies. In complete handoffs, information about *all* objects accessed by the client is transferred, while in partial handoffs the recipient proxy is told about only a fraction of the objects accessed by the client (e.g., the most popular $k$ objects or the most recently accessed $k$ objects).

Thus, depending on whether information about some or all objects is transferred and whether only meta-data information or both meta-data and data are transferred, we get four different combinations of handoff mechanisms. In the next two sections, we present actual handoff algorithms that instantiate these concepts for push and pull.

### 2.3 Handling Disconnections in Mobile Environments

Since mobile clients can get disconnected at any time in wireless environments, timely detection of disconnections and reconnections is essential for maintaining coherency guarantees. As explained earlier, the proxy and the server continue to be connected to one another during client disconnections. Consequently, the proxy continues to receive updates to dynamic data items during this period; these updates are received either using server-push or proxy-pull. Due to the disconnection, however, these updates can not be propagated to the client and must be buffered at the proxy. We assume that the proxy employs a circular buffer of size $N$, $N > 1$, for each dynamic data item. Thus, up to $N$ most recent updates can be buffered during the disconnection; these buffered updates are propagated to the client upon reconnection.

To ensure such buffering of updates and timely resynchronization, both the proxy and the client need to detect disconnections and reconnections in a timely fashion. Detecting such events at the client is simple; most commonly used wireless network interface cards provide connectivity information that indicates whether network converge is available at a certain location (e.g., mobile phones have a signal strength indicator indicating if a signal is present and the signal strength; drivers for 802.11b wireless also provide signal strength information to the operating system). Consequently, the end-client can determine its connectivity status by simply querying the network interface card, and detection of disconnections and reconnections is immediate. Upon detecting a disconnection, the client can service application requests for web objects from its local cache for a maximum period $\Delta$ (as allowed by time-domain $\Delta_e$ consistency). To prevent violation of coherency guarantees, cached items are marked as potentially stale $\Delta$ time units after a disconnection; no further application requests can be serviced beyond this point until reconnection.

Detecting client disconnections and reconnections at the proxy is more challenging. We assume that proxy resorts to the use of heartbeat messages to determine if a client is connected. Such heartbeat messages are sent periodically to the client; the lack of a response is assumed to signal a disconnection. Similarly, heartbeat messages are sent periodically after a disconnection to detect a reconnection; a subsequent client response indicates a reconnection.

The frequency of heartbeat messages governs the timeliness of detecting network connectivity events. The more frequent these messages, the better the accuracy; on an average, the proxy incurs a time lag of $\tau/2$ for detecting disconnections and reconnections, where $\tau$ is the interval between successive heartbeat messages. Note that, any interaction with the client (e.g., a request for a web page, propagation of an update) serves as an implicit heartbeat (and can be used reduce the frequency of explicit heartbeats). Such implicit heartbeats can be used to improve the accuracy of detecting both disconnections and reconnections. We assume that a disconnection causes the proxy to start buffering updates on behalf of the client, which are then sent to the client upon reconnection.

## 3 Push-based Dissemination in Mobile Environments

In this section, we consider how techniques for handling intermittent connectivity and client mobility can be adapted for push-based dissemination. We first describe how push-based data dissemination operates in mobile environments and then describe our handoff algorithm.

### 3.1 Maintaining $\Delta_e$ Consistency Using Push

In push-based dissemination, we assume that a client registers each data item of interest with the proxy and specifies a coherency tolerance $\Delta$ for each data item. The proxy in turn registers these data items with the corresponding servers along with the corresponding coherency tolerances. The server then tracks changes to each such data item and pushes all updates that exceed the tolerance $\Delta$ to the proxy. The proxy in turn pushes these updates to the client. In the event the client is disconnected, the updates are buffered and propagated to the client upon reconnection.

Since the server maintains state information to determine which updates should be pushed, we assume that *leases* are employed to limit the duration for which such updates are propagated. A lease requires the server to propagate updates only for a fixed (lease) duration—the lease must be explicitly renewed upon expiration if further updates are desired (the state information is discarded if the lease is not renewed and no further updates are pushed).

Thus, the proxy and the server are *stateful* in the push approach approach. The specific state information maintained at the server consists of the object ID, proxy ID, the lease duration and the tolerance $\Delta$. The state information maintained at the proxy consists of client-specific state such as the list of all client-requested objects as well as object specific state such as the object ID, lease expiration time, last update received, tolerance $\Delta$ and up to $N$ buffered updates.

### 3.2 Handoff Algorithms for Push

Assuming the above environment, the handoff algorithm employed for push-based dissemination of dynamic data is outlined in Figure 2(a). As shown in the figure, the process begins when proxy $P_1$ initiates a handoff with proxy $P_2$. Assuming that $P_2$ has the resources to service the client, $P_1$ then transfer client-specific state for client $C$ to proxy $P_2$. The figure outlines the information that is transferred during complete optimistic handoffs—both meta-data and the buffered data is transferred for all client-requested objects. As explained earlier, only a subset of this information may be transferred for partial or pessimistic handoffs. Once the state information has been transferred, the proxy requests the server to send a copy of all subsequent updates to $P_2$. At this point, both $P_1$ and $P_2$ begin receiving updates to all items with valid leases. Since the client $C$ is completely oblivious of the message exchanges thus far, only updates received by $P_1$ are forwarded to the $C$ and $P_2$ buffers all updates received from the server. (thus, $C$ continues to interact with $P_1$ in the normal fashion until this point). Beyond this step, $P_1$ suspends propagating updates and initiates a transfer of responsibilities to $P_2$. This is done by sending a message to the client asking it to send all subsequent requests to $P_2$. Upon receiving an acknowledgment, $P_1$ requests $P_2$ to forward all updates received since the suspension of update propagation to the client (i.e., all updates received during step 7 are sent). This is achieved by sending the sequence numbers of all updates that have already been propagated to $C$ and $P_2$ simply propagates all buffered updates following this sequence number. Proxy $P_1$ then requests the server to stop sending it updates and thereby transfers all active leases to proxy $P_2$. At this point the handoff process is complete and $P_2$ takes over responsibilities for servicing the client.

To ensure correctness, the handoff process must handle the scenario where the client gets disconnected before the handoff is complete; To do so, proxy $P_1$ records the transfer of responsibility to $P_2$ and simply asks the client to switch to $P_2$ upon reconnection (step 7). It can be shown that the above handoff algorithm has the following key property: *No updates are missed by the client as a result of the handoff.* This property ensures that the handoff process is seamless and causes minimal disruption to the client.

**Property:** *No updates are missed by the client as a result of handoff in the push–based data dissemination approach.*

**Proof:** Consider the same scenario as above, client $C$ sending requests via proxy $P_1$ and $P_1$ initiating a handoff with proxy $P_2$ due to client movement.

- Before the handoff algorithm is intitiated, $P_1$ forwards notfications of updates objects to $C$.
- $P_2$ starts receiving notifications for updated objects at the server after Step 6 of the handoff algorithm. Even at this stage, $P_1$ is still responsible for forwarding notifications to $C$.
- After Step 8 of the handoff algorithm, the client has either updated is proxy mapping to $P_2$ or $P_1$ has recorded the handoff information as $C$ is disconnected. $P_1$ sends a take–over message to $P_2$ with sequence numbers of objects used in handoff. At this point $P_1$ stops forwarding notifications to the $C$.
- $P_2$ takes–over the responisibility of forwarding notifications of updated objects to $C$. As the first step, $P_2$ uses the sequence numbers of objects and the consistency parameter '$\Delta$' to forward pending notifications to $C$ during execution of step 8.

As a result, with the use of sequence numbers on objects no updates are missed by the client as a result of handoff.

### 3.3 Detecting client disconnections and reconnections

The techniques discussed in Section 2.3 can be employed for detecting disconnections and subsequent reconnections at the client and the proxy. As explained earlier, the proxy buffers updates received from the server during a disconnection and propagates them to the client upon reconnection. Since no special techniques are required for the push-environment, we do not discuss this issue any further.

## 4 Pull-based Dissemination in Mobile Environments

In this section, we discuss techniques for enforcing $\Delta_e$ consistency using pull-based techniques. We first provide an overview of the pull-based approach for disseminating dynamic data and then discuss handoff algorithms and techniques for handling mobility.

### 4.1 Maintaining $\Delta_e$ Consistency Using Pull

In the pull-based approach, we assume that the mobile client registers data items of interest with the proxy and also specifies the desired coherency tolerance $\Delta$ on each data item. The proxy them polls the server periodically to see if

1. $P_1 \rightarrow P_2$: initiating handoff for C
2. $P_2 \rightarrow P_1$: ACK
3. $P_1 \rightarrow P_2$: C needs $\{< d_1, \Delta_1, S(d_1) >, < d_2, \Delta_2, S(d_2) >, \ldots\}$
4. $P_2 \rightarrow P_1$: ACK
5. $P_1 \rightarrow S$: Send updates for object in C to $P_2$
6. $S \rightarrow P_1$: ACK
7. `if` $C$ `is connected`
   i. $P_1 \rightarrow C$: Switch to $P_2$
   ii. $C \rightarrow P_1$: ACK
   `else if` $C$ `is disconnected`
   Update record of handoff for $C$ at $P_1$
8. $P_1 \rightarrow P_2$: Take-over client $C$ $\{< d_1, S(d_1) >, < d_2, S(d_2) >, \ldots\}$
9. $P_2 \rightarrow P_1$: ACK
10. $P_1 \rightarrow S$: Update state of objects in $C$ transferred from $P_1$
11. $S \rightarrow P_1$: ACK

(a) Handoff algorithm for push

1. $P_1 \rightarrow P_2$: initiating handoff for C
2. $P_2 \rightarrow P_1$: ACK
3. $P_1 \rightarrow P_2$: C needs $\{< d_1, ttr_1, S(d_1) >, < d_2, ttr_2, S(d_2) >, \ldots\}$
4. $P_2 \rightarrow P_1$: ACK
5. `if` $C$ `is connected`
   i. $P_1 \rightarrow C$: Switch to $P_2$
   ii. $C \rightarrow P_1$: ACK
   `else if` $C$ `is disconnected`
   Update record of handoff for $C$ at $P_1$
6. $P_1 \rightarrow P_2$: Take-over client $C$ $\{< d_1, S(d_1) >, < d_2, S(d_2) >, \ldots\}$
7. $P_2 \rightarrow P_1$: ACK

(b) Handoff algorithm for pull

**Figure 2**: Handoff algorithms

each data item has changed and if so, fetches (and buffers) a new version of the data item. To meet client-specified coherency guarantees, the proxy must poll the server at least once every $\Delta$ time units (which ensures that the version at the proxy and that at the server are never out of date by more than $\Delta$ time units). The time periods between successive polls is determined using a per-object parameter referred to as *time-to-refresh (TTR)*; thus $TTR = \Delta$ in our approach. It is also possible to compute the TTR value dynamically based on the rate of change of the object (instead of using a static value of $\Delta$). In this dynamic TTR approach, the proxy uses more conservative values of TTR for more frequently changing objects and uses larger values for infrequently changing objects (thereby reducing the poll overhead for infrequently changing objects). Such dynamic TTR techniques have been studied in [8] and can be employed in mobile environments as well.

Regardless of whether the TTR is computed statically or dynamically, the proxy buffers all updates fetched from the server (unlike the push approach, where such updates are pushed immediately to the client). We assume that a similar pull-based approach is used on the proxy-client path. The client is assumed to periodically poll the proxy and all updates buffered by the proxy since the previous poll are then delivered to the client. In general, the polling frequency on the client-proxy path can be different from the one employed on the proxy-server path. Further, the client can also use a dynamic TTR technique to reduce the overhead of unnecessary polls. For simplicity, in thus paper, we assume that the client polls the proxy once every $\Delta$ time units when connected. No polls are possible during a disconnection and all updates received by the proxy during this period are delivered upon a reconnection.

Observe that in the pull approach, only the proxy is stateful, while the server is stateless. Consequently, the pull approach can be implemented in the context of the HTTP protocol, unlike push, where additional mechanisms are necessary for maintaining state and pushing updates.

### 4.2 Handoff Algorithm for Pull

Figure 2(b) describes the basic handoff algorithm for the pull approach. Since the server is stateless in the pull approach, the handoff algorithm is simpler than that in the push approach. Like in push-based handoffs, $P_1$ initiates a handoff by contacting $P_2$. Assuming $P_2$ has sufficient resources to service a new client, $P_1$ then transfers client-specific state for $C$ to proxy $P_2$. The algorithm in Figure 2(b) indicates the state information that is transferred for complete optimistic handoffs (in this case, the object ID, the TTR, the last update received and the contents of the buffer are transferred). Only a subset of this information needs to be transferred if the handoff is pessimistic or partial. Once this state information is transferred, proxy $P_2$ has sufficient information to service the client. At this point $P_2$ also begins polling the server for each data item of interest to the client. The proxy $P_1$ then simply transfers responsibilities to $P_2$ by asking the client to use $P_2$ for all subsequent polls. An polls received by the proxy while this transfer is in progress are simply redirected to proxy $P_2$ (using the HTTP redirect mechanism); $P_2$ services these requests once the transfer is complete. Note that, unlike push-based handoffs, no interactions with the server are necessary, which simplifies the handoff process. Like in push-based handoffs, our pull-based handoff algorithm can also guarantee that no updates will be missed by the client. This ensures that the handoff is seamless to the client.

**Property:** *No updates are missed by the client as a result of handoff in the pull–based data dissemination approach.*

**Assumption 1:** $TTR_{client-proxy} \leq MinTTR_{proxy-server}$, the TTR for an object at the client is less than or equal to the minimun TTR for objects at the proxy.

**Assumption 2:** Clients are always connected to the proxies.

**Proof:** Consider the same scenario as above, client $C$ sending requests via proxy $P_1$ and $P_1$ initiating a handoff with proxy $P_2$ due to client movement.

- $C$ polls $P_1$ to get versions of objects updated at the server, before handoff is initiated.
- After step 4 of the handoff algorithm, $P_2$ has the objects participating in the handoff and their corresponding *ttr* information. At this point, $P_2$ has enough information about the objects and starts polling the server for updates based on the *ttr* information. The client $C$ is still polling $P_1$ for modified objects.
- After step 5, $C$ polls $P_2$ for modified objects and $P_1$ stops polling the server if required.
- $C$ polls $P_1$ before step 5 and $P_2$ after step 5. $P_2$ has enough information before step 5 to get the same versions of objects as $P_1$ before step 5.

As a result, no updates to objects are missed by the client as a result of handoff.

**Note:** If we do not make Assumption 1 as stated above, then the proxy can hold more than one versions for the object during a handoff. Since our handoff algorithm transfers the most recent version of the object and not all versions, updates can be lost at the proxy.

### 4.3 Detecting client disconnections and reconnections

As before, the techniques discussed in Section 2.3 can be employed for detecting disconnections and subsequent reconnections at the client. The requirements at the proxy are even simpler—since the client is responsible for pulling updates from the proxy, the proxy need not track whether a client is connected or disconnected. The proxy simply continues to poll the server and buffers all updates—it is the responsibility of the client to pull these updates. If the client is connected, it pulls these updates within $\Delta$ time units; if disconnected, it pull these updates upon reconnection. The proxy does not distinguish between the two scenarios and hence, does not need to track the connectivity status of the client. Note that, the client still needs to determine its connectivity status (by querying its network interface) and must mark all objects as possibly stale $\Delta$ time units after a disconnection. Similarly, it must resynchronize the state of all stale objects upon reconnection by polling the proxy.

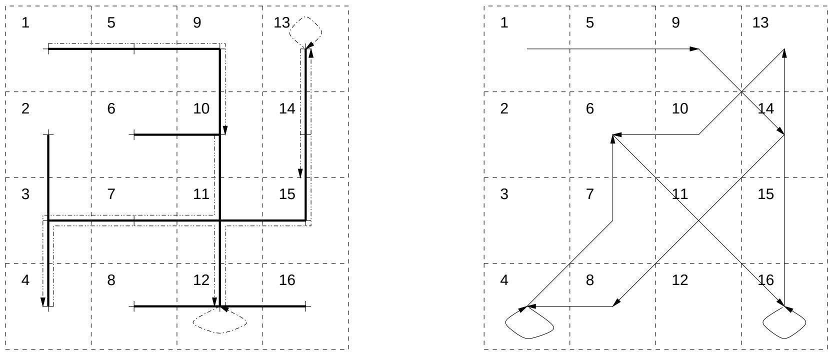
## 5 Mobility models

In order to test and verify the efficacy of our solution we need to use mobility models that represent the movement characteristics of mobile hosts. A mobility model defines characteristics of movement like speed, direction for a mobile host. The choice of a model is important, as it should closely reflect the characteristics of an actual environment for the simulation experiments. We use two mobility models, the Fixed-Path Model and a variation of the Random Way Point Model, to model movement characteristics of hosts for the experimental evaluation.

### 5.1 Fixed-Path Mobility Model

The Fixed-Path mobility mobility model is used to represent environments where mobile hosts can move from one location to another using a fixed set of paths. Typical examples of such environments are movements of mobile hosts in a city, where hosts move only along streets and not in any arbitrary direction to reach a destination or an university campus where hosts move to different destinations on the campus along fixed paths. The model can also be used to represent movement from one city to another using a fixed set of streets and roads to the destination. In many such environments, hosts cannot move in any random direction or through obstacles, but can move only along fixed directions and paths. The important feature of this model is restricting movement of mobile hosts by using a fixed set of connection/movement paths.

We have instantiated the model in the following manner. The simulation region consists of a number of cells and wireless hosts communicate via a base station serving that cell. Each cell can be abstract as a block in a city or a city itself or a part of the university campus. Cells are connected to each other with fixed paths, i.e.: a host in a cell can move to another cell only if there exists a path between them. The cells can be viewed as nodes in a graph and paths between cells as edges connecting the nodes. We assume bi-directional path between cells. A mobile host either is moving or is pausing at a cell. A mobile host decides with equal probability whether it should pause at its current location or move to another location. If the host decides to pause, it chooses a pause time from a uniform random distribution bounded by a maximum period. If the host decides to move, it selects a location(cell) at random and finds the shortest path via other cells to the destination. The host starts moving towards the destination using the shortest path discovered with constant speed. [1] Another assumption we make is that movement from a cell always starts from the centerer of the cell, i.e.: the location of a host is a cell and not a particular location in the cell. We also assign a size to the cell to calculate time spent to in a cell while moving.




(a) Fixed–Path (b) Variation of Random–Way Point

**Figure 3**: An environment with 16 cells and showing movement of a host starting at cell 1.

Figure 3(a) shows an example environment with 16 equal sized cells. The valid paths between cells are shown in thick lines and movement of a host starting from cell 1 is illustrated using dashed lines and directed arrows. Initially, a host in cell 1 decides to moves and chooses cell 10 as its location. The shortest path calculated from the available paths is $1 \rightarrow 5 \rightarrow 9 \rightarrow 10$. After reaching the destination, it decides to move again and selects another location and continues. The loops in cell 12 means the host decide to pause at the cell and stayed there for a pause time duration, before he made another decision to move or pause.

We do not restrict when a host can make a request for a web object, the host can send a request while its moving or is stationary as long as it is connected to the base station.

[1]The implementation can be extended to associate different speeds to different paths. This is more realistic as streets in the center of the city may be crowded and ones in the suburbs maybe relatively free.

### 5.2 Variation of Random Way Point Model

The Random Way Point model is similar to the Fixed-Path model but more flexible in how hosts can move from one location to another. As in the previous case each host can either pause at a cell or choose another destination cell and start moving towards it. The model does not restrict movement along fixed paths. When a host decides to move, it chooses a destination cell and as a result a direction to move towards the destination. The host then moves along a straight line in that direction with a chosen speed. In the original model, the host can may in any direction and does not take into consideration if actual paths exist. The original model is developed for mobile hosts in an ad-hoc network where hosts can move arbitrarily in any direction to reach the destination.

We use a variation of the model which controls the movement to a slightly higher degree. As before our simulation environment is divided into cells. A host is identified by a location/cell and always starts movement from the center of the cell. Once a host chooses a destination cell to move, it does not move in any direction. Instead, the host moves at a 45 degrees angle along the direction of the destination to a reach an intermediate cell via which the destination can be reached along a horizontal or vertical direction.

Figure 3(b) illustrates movement of a host in and 16–cell grid using the variation of the Random Way Point model. The host is initially in cell 1 and decides to move to cell 9 and moves in a straight line with a constant speed. Speed of movement for our experiments is constant but can be chosen at random for each different movement periods. Consider movement between source cell 13 and destination cell 6 of the host. In this case the destination cannot be reached in a straight line directly. The host first moves along a 45 degree angle towards the direction of the destination to reach cell 10 and then moves in a straight line from cell 10 to destination cell 6. As in the previous case loops as in cell 4 and 16 depict a pause by the host in that cell.

Movement across cell is always through the center of the cell along the diameter or the paths running parallel to the sides. This is not similar to the original random way point model where hosts can move in any direction. But as the random way point model was meant for ad-hoc networks there were no cells associated with cells and absolute locations of hosts served as location of hosts. But in our case, we have a cell as a hosts location and the variation makes distance and movement calculations simpler and also maintains the original random movement idea.

## 6 Experimental Evaluation

In this section, we evaluate the efficacy of our approaches using simulations. Our experiments quantify and compare the impact of various mobility and disconnection parameters on push-and pull-based dissemination.

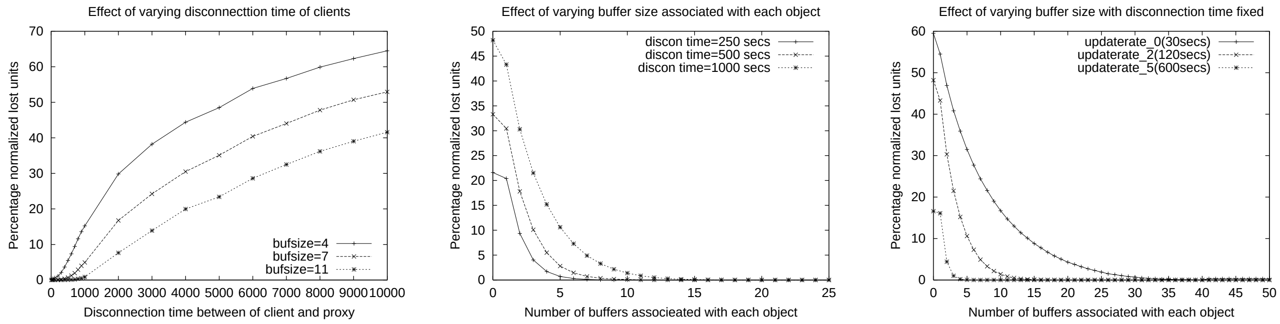
### 6.1 Experimental Methodology

**Mobility Model**: We use the Fixed-Path mobility model to model client mobility (i.e., the movement characteristics of mobile hosts).

**Simulation Environment**: The simulation environment consists of a number of cells and wireless mobile hosts which communicate via base stations. Cells are connected to each other with fixed paths, i.e.: a host in a cell can move to another cell only if there exists a path between them. We assume bi-directional paths between cells. Clients can send requests for data so long as they are connected to a base station. An ON–OFF process simulates the periods for which a client is connected to the base-station and periods for which it is disconnected. Every client toggles between the two states and connection and disconnection periods are bounded by a maximum. In our experiments, we keep the connection period fixed and each disconnection period is chosen at random from a uniform distribution bounded by a maximum. The environment has a set of proxies that are associated with cells using a simple mapping function. A proxy can be associated with more than one cell, which implies that requests from clients in more than one cell are forwarded to the same proxy. We assume each proxy maintains a disk-based cache and has infinite size. On client disconnections, the proxy can also buffer multiple versions of objects that are served to the client after reconnection. The clients also have a local cache to store objects and we assume the client caches to be of finite size.

| Trace | Approx. Duration | Number of requests | Unique requests | Number of updates |
|---|---|---|---|---|
| DEC | 24 hrs | 1000000 | 354497 | 778327 |
| SYNTHETIC | 4 hrs | 977000 | 100000 | 924215 |

**Table 1**: Workload characteristics




(a) Maximum disconnection period (b) Number of buffers per object (c) Average update rate at server

**Figure 4**: Effect of intermittent connectivity in push-based data dissemination (DEC trace).

If a client is about to make a request and detects that it is not connected to the base-station, that request is dropped and the simulator moves to the next request in the trace.

**Workload Characteristics**: The workload for our experiments is generated using traces containing several requests. Each request contains information about the client, object identifier, size, time of request etc. The simulator also appends to each request the current location of the client according to the mobility model[2]. We report results for experiments conducted with the publicly available DEC proxy trace and a synthetically generated trace using WebTraff – A Proxy Workload Generator [1].

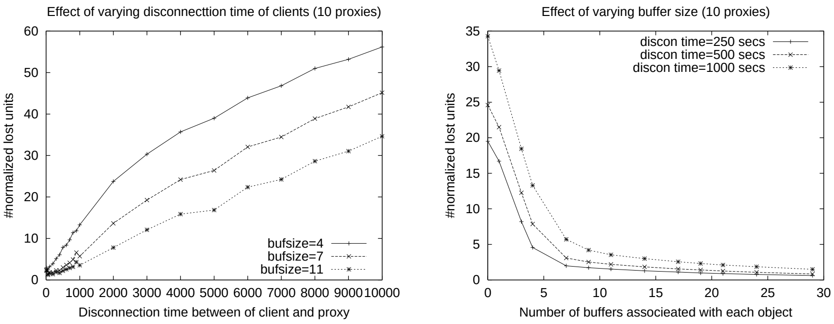
We also simulate updates to objects at the server using an update process. The objects present at the server are divided into two categories, *mutable* and *slowly–mutable*. Approximately 50% unique objects are mutable and the other half slowly-mutable. We perform experiments by varying the average interval between updates for mutable objects. As a default case, the 50% mutable objects are updated approximately every 2 minutes and the slowly-mutable objects are updated every 8 hours.

We use a 24 hour portion of the DEC trace with a million requests, 354497 unique objects and 778327 updates to objects at the server. The synthetic trace has 977000 requests, 100000 unique objects and 924215 updates to objects at the server. The trace characteristics are summarized in Table 1.

## 6.2 Effect of intermittent connectivity of clients

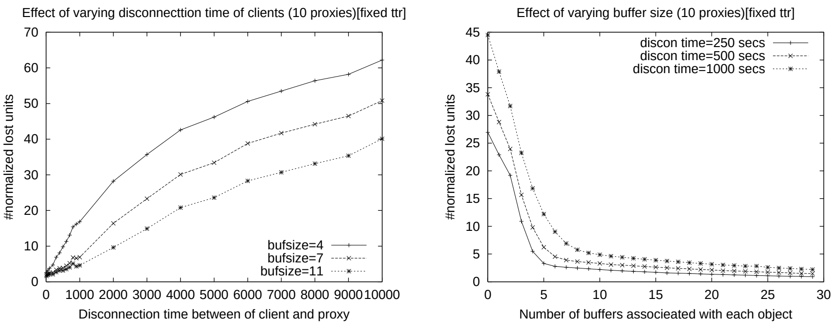
To study the effect of intermittent connectivity of clients, we varied three different parameters, one at a time: the maximum disconnection time of clients, the size of the per-object circular buffer and the average time between updates of mutable objects at the server. As the dependent variable, we measured the percentage of *normalized lost units*. Each new update is assumed to be appended to the circular buffer and may overwrite the least recently received update in the buffer (which is assumed to be lost). The normalized lost units metric is defined to be the total number of lost updates normalized by the number of updates sent by the server. Since we are quantifying the

[2]We assume that each client knows its current location when it makes a request or a proxy can track its location based on information from the lower–levels of the protocol stack.




(a) Maximum disconnection period (b) Number of buffers per object

**Figure 5**: Pull with dynamic TTR (DEC trace).



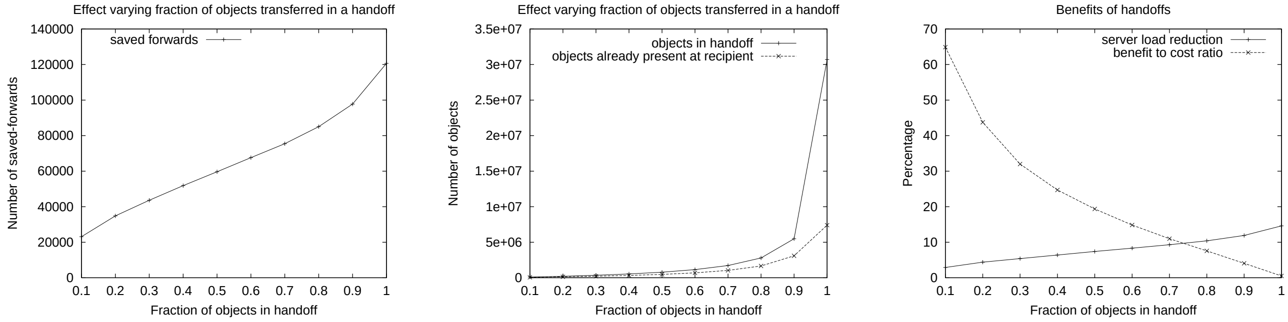

(a) Maximum disconnection period (b) Number of buffers per object

**Figure 6**: Pull with static TTR (DEC trace).

effect of intermittent connectivity, all clients are assumed to be stationary in this experiment. The experiments are conducted with 100 clients, 10 proxies and connection period of 500 seconds in a push-based data dissemination approach.

Figure 4(a) shows the effect of varying the maximum disconnection duration on lost updates. The figure shows that the normalized loss increases with increase in disconnection period (since larger disconnections increase the probability of an update being overwritten in the buffer). Figure 4(b) shows the impact of varying the size of the circular buffer on loss. Not surprisingly, larger buffers yield a smaller loss. We see that, for a given disconnection time, the loss curve has a "knee" beyond which the loss percentage is small. Choosing a buffer size that lies beyond the knee ensures that the proxy can effectively mask the effect of a disconnection (by delivering all updates received during such periods). In general, we find that a small number of buffers ($15-20$) seem to be sufficient to handle disconnections as large as 15 minutes. As few as 5 buffers can reduce the loss rate from 33% to 2.8% for disconnections of up to 500 seconds. Lastly, we vary the write frequency (update rate) of the objects and examine the its impact on loss; the maximum disconnection period is set to 1000 seconds. Figure 4(c) shows that the normalized loss increases with the update rate (since more frequent updates increases the chances of overwriting a prior update in the circular buffer). Even for very frequent updates (once every 30 seconds) and disconnections as large as 1000 seconds, we find that buffer sizes of 25-30 are sufficient to mask the effects of disconnections (by incurring very little or no loss).

While the above experiments are for the push scenario, we observed similar trends for pull as well. Figures 5 and 6 report results for a pull–based approach with dynamic–TTR and static–TTR respectively. Figures 5(a) and 6(a), show that in both cases, with increase is maximum disconnection period of clients, the normalized loss of buffer units at




(a) Saved forwards (b) Objects in handoff (c) Benefits and feasibility of handoff

**Figure 7**: Effect of client mobility in push-based data dissemination (DEC trace).

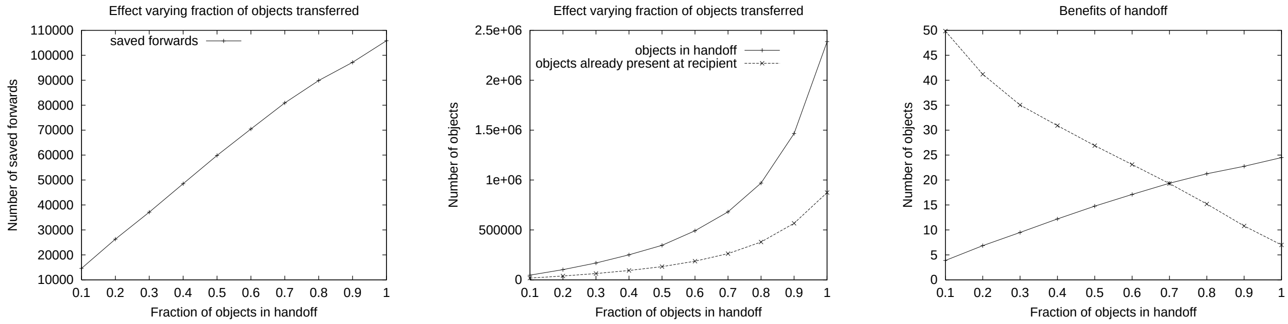
the proxies increases. Figures 5(b) and 6(b) show the impact of increase in the number of buffers associated with each object at the proxy. As in the push–based approach, larger buffers yield smaller loss and the loss curve has a “knee” beyond which loss percentages are small.

### 6.3 Effect of client mobility

Next we study the impact of client mobility. Since we are interested in mobility issues, our experiments assume that client are always connected (i.e., no disconnections). We study handoff algorithms for push and pull by varying the fraction of objects transferred in each handoff and measuring various overheads. The experiments are performed using 100 cells, 10 proxies and 10 clients and a lease duration of 30 minutes for the push-based approach.

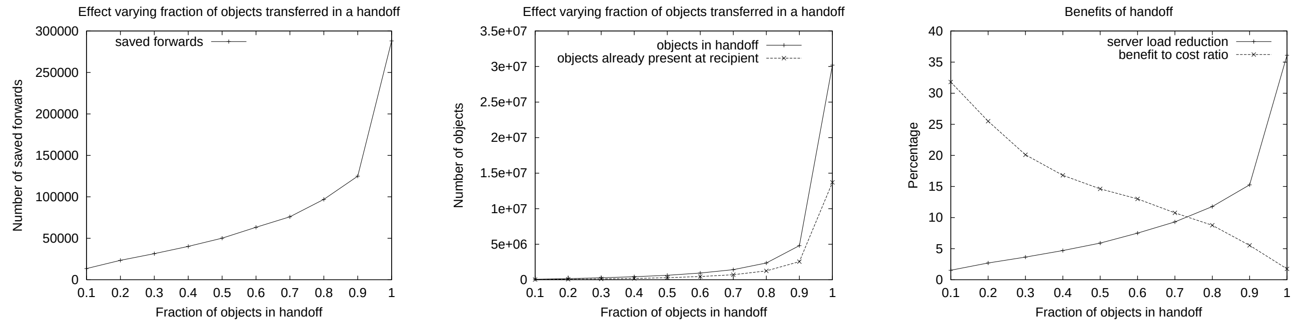
Figure 7(a) shows the number of client accesses that were hits at proxies due to handoffs. The metric *saved–forwards* is the number of number of client requests served from the proxy cache after a handoff (which saves on the overhead of going to the server if this object had not been transferred). We find that, the greater the fraction of client-specific state transferred, the larger are the resulting savings (since a larger number of subsequent client requests will be cache hits).

The total number of objects transferred in handoffs over the entire simulation and number of those objects that are already present at the recipient proxies are plotted in Figure 7(b). Transferring an object that is already present at the recipient proxy results in a wasted overhead; an intelligent handoff scheme will transfer only objects that are not present at the recipient proxy. We find that the greater the fraction of client-specific state transferred, the greater




(a) Saved forwards (b) Objects in handoff (c) Benefits and feasibility of handoff

**Figure 8**: Effect of client mobility in push-based data dissemination (Synthetic trace).




(a) Saved forwards (b) Objects in handoff (c) Benefits and feasibility of handoff

**Figure 9**: Effect of client mobility in pull-based data dissemination with dynamic TTR (DEC trace).

are the chances that an object will be already present in the recipient proxy cache for subsequent handoffs. This indicates that a complete handoff algorithm should be selective in the the objects it transfers, since some of those object may already be present at the recipient and need not be resent.

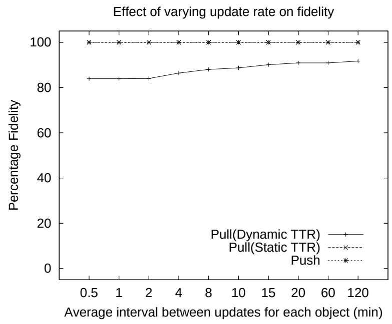
Figure 7(c) quantifies the benefits of handoffs in two different ways. The first metric is the reduction in server load due to handoffs (defined as the number of saved forwards normalized by the total number of server requests and saved forwards). As can be seen, the larger the amount of client state transferred, the larger is the reduction in server load (since this results in more hits at the recipient proxy cache). We observe a 13% reduction in server load for complete handoffs. Our second metric is the benefit to cost ratio for handoffs and is defined as ratio of the reduction in server load (i.e., number of saved forwards) and the handoff message overhead. We find that the larger the amount of client-state transferred, the greater the benefits and greater the cost. As shown in the figure, the cost increases faster than the benefits, resulting in smaller benefit to cost ratios. The ratio falls from 65% to 2% when the fraction of client state transferred is increased from 10% to 100%. Together, these two metrics indicate that the fraction of client state transferred should be chosen judiciously to balance the reduction in server load with the cost of achieving that reduction (i.e., handoff overhead). Note that there is an additional benefit of handoffs, namely in client response times, that is not quantified here.

We repeated the experiment with a synthetic trace and observed similar results — Figure 8 reports these results. With complete handoffs we see a 24.5% reduction at the server load, i.e., 105844 saved–forwards. With 30% of the objects associated with the client transferred on each handoff, we observe a server load reduction of 9.5%. Similar to the previous experiment, as the fraction of objects transferred in each handoff increases, we see increased benefits but at a greater cost. The benefit to cost ratio for 30% objects transferred is 35.04% and varies from a maximum of 49.9% to a minimum of 7%.

We also observed similar trends for experiments conducted with a pull–based approach using dynamic TTR and the DEC proxy trace. The results of these experiments are depicted in Figure 9. As in the previous two experiments, we see increase in benefits with increased fraction of objects transferred in each handoff and also increase in cost. With complete handoffs, we observe a 31.8% reduction in server load, i.e., 288051 saved–forwards to the server. The benefit to cost ratio varies from 31.8% to 1.75% as the fraction of objects in handoff is varied from 10% to 100%. The pull–based approach resulted in more *saved–forwards* for higher fractions than the push-based approach. We attribute this to the large TTR values at the proxy, due to which objects are considered *fresh* for longer periods than the corresponding lease duration in the case of the push–based approach.

### 6.4 Comparison of Push and Pull

In this section we compare the overheads of push and pull for handling intermittent connectivity and client disconnections.




(a) Fidelity of objects at clients

| Max. discon. time | Push | Pull (Dynamic TTR) | Pull (Static TTR) |
|---|---|---|---|
| 250 secs | 612,299 | 6,118,343 | 268,634,478 |
| 500 secs | 491,273 | 5,105,410 | 228,683,276 |
| 1000 secs | 357,164 | 3,981,911 | 181,113,610 |

(b) Message overhead for consistency

| Fraction of objects in handoff | Pull | Push |
|---|---|---|
| 0.1 | 5.34 | 7.14 |
| 0.2 | 9.25 | 11.02 |
| 0.3 | 14.35 | 16 |
| 0.4 | 20.65 | 22.5 |
| 0.5 | 28.82 | 31.15 |
| 0.6 | 40 | 44.15 |
| 0.7 | 57.14 | 64.43 |
| 0.8 | 87.8 | 102.94 |
| 0.9 | 174 | 215.64 |
| 1 | 1323.75 | 2057.85 |

(c) Handoff overhead

**Figure 10**: Comparison of push and pull based dissemination approaches.

Figure 10(a) plots the fidelity of the data for push and pull in the presence of client disconnections. As expected, the push-based approach and in case of static TTR in pull (poll once every $\Delta$ time units) yields perfect (100%) fidelity (since cached data is always uptodate with the server). The dynamic TTR approach, which uses intelligent polling based on the observed rate of change of objects, yields $83.9 - 91.7\%$ fidelity for update rates varying from once every 30 seconds to once every 2 hours. The number of server-proxy messages incurred in these three scenarios is reported in Figure 10(b). As shown, the push-based approach has the least message overhead (since the server pushes an update only when necessary). The static TTR approach incurs the largest overhead, since many of its (frequent) polls are unnecessary. The dynamic TTR approach incurs an overhead that is larger than push but smaller than the static TTR; however, this is at the expense of lower fidelity, since some updates are missed by the polling technique. *We conclude that a push-based approach is better suited for coherent data dissemination in the presence of disconnections due to its lower message overhead and 100% fidelity (but at the expense of making the servers stateful).*

Next, we examine the average message overhead per handoff for push and pull (see Figure 10(c)). We see that pull has a lower handoff overhead than push regardless of the fraction of client-state transferred. This is because (i) there are no interactions with the server during a pull-based handoff and (ii) objects expire in the pull approach only when the TTR expires— as a result, cached objects are *valid* for longer durations in pull than in the push with leases (this reduces the number of objects that must be transfered since more of these objects are likely to be present at the recipient). Consequently, we find that *a pull-based approach is better suited for handling client mobility than push due to its lower handoff overhead.*

## 6.5 Detecting reconnections at the proxy

As discussed in Section 2.3, the proxy needs to detect disconnections to maintain temporary state and detect reconnections to forward state to clients. This is an important in the push–based approach, as the proxy is responsible for coherency of objects served by the clients. The proxy employs a heartbeat mechanism to detect reconnections and a disconnection is detected when a proxy message does not reach the client.[3] In our scheme, the proxy periodically sent heartbeat messages to the client to check connectivity. Client requests serve as implicit heartbeats, but in our experiment the detection process always polled the client at fixed intervals.[4]

We performed an experiment to detect the average time to discover a reconnection of clients at the proxy with different intervals for heartbeat messages. As seen from Figure 11, the average time to detect a reconnection increases with increase in heartbeat intervals. The time to detect reconnection also increases with increase in the maximum

[3]Alternatively, the client can also send an exclusive resynchronize message to the proxy on a reconnection.

[4]As an alternative each client request should be treated as a heartbeat message and the next poll calculated on the arrival time of the request.

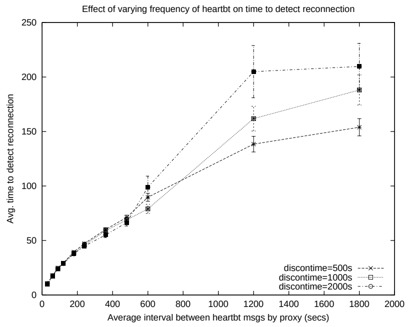



**Figure 11**: Vary heartbeat interval to measure time to detect reconnection at proxy (DEC trace).

disconnection period of clients. As client requests to proxies act as implicit heartbeats, the average time for detection is similar for smaller heartbeat intervals.

## 7 Related work

File systems such as CODA [12, 14] have investigated disconnected operations for mobile clients and techniques for hoarding files. Other distributed systems like Bayou [9] have also looked at issues in maintaining consistency in weakly connected systems. These systems assume that updates can be made at any client (or replica) and use epidemic protocols to efficiently propagate these updates to all other hosts or to a central server in the presence of weak connectivity. In contrast, our solution is specific to the web (where updates are made only at the server) and we focus on the efficient dissemination of these updates from server to client via proxies (in the presence of intermittent connectivity and mobility).

There have been various efforts on maintaining consistency of data cached at mobile clients. [5] and [15] propose techniques to propagate invalidation reports on reconnections and purging of client cache on disconnection. We extend the idea to provide $\Delta_e$–consistency for data at the clients. Our approach also uses buffers at the proxies to store partial state of client data during disconnections. A solution where mobile hosts use the mobile switching stations as home locations to store state during disconnections is proposed in [11]. Mobile clients inform their respective home locations of each access to update current state. The state of hosts is moved across stations as hosts change locations. Or approach of handling mobility is similar, but we extend the idea to store state at proxies and move client state across proxies. A solution for disseminating data over asymmetric wireless networks is proposed in [2]. The approach is based on periodic broadcast of data, where the entire dataset is replicated and prioritized for broadcast. Infostation–based downloads [10] is another broadcast based data dissemination scheme. We do not assume an asymmetric channel or a replicated dataset or a broadcast based dissemination scheme, Our solution involved proxies used for caching in a request–response mechanism of data access.

While the previous efforts are for disseminating static web data and for reconciling changes at clients with other hosts on reconnection, our solution is better suited for dynamic data that changes frequently and disseminating data from servers to mobile clients.

## 8 Conclusion

In this paper, we argued that the intermittent connectivity and client mobility of wireless devices require a reexamination of push and pull-based techniques for disseminating dynamic data. To address this issue, we introduces a new coherency semantics called *eventual-delta* consistency for mobile environments. We then showed how push and pull-based dissemination techniques should be adapted to (i) exploit client mobility and (ii) provide temporal coherency guarantees even in the presence of client disconnections. Specifically, we proposed (i) proxy-based buffering techniques to mask the effects of client disconnections and (ii) application-level handoff algorithms to handle client mobility. Our experimental evaluation demonstrated that push is better suited to handle client disconnections due to its lower message overhead and better fidelity, while pull is better suited to handle client mobility due to its lower handoff overheads.

## References


[1] WebTraff – GUI for Web Proxy Cache Workload Modeling, Analysis, and Simulation. http://pages.cpsc.ucalgary.ca/ carey/software.htm.

[2] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *ACM SIGMOD*, pages 183–194, 1997.

[3] Anonymous. Handling Client Mobility and Intermittent Connectivity in Mobile Web Access. Technical report, Anonymous Institution, 2002.

[4] P. Bahl and V. N. Padmanabhan. RADAR: An In-Building RF-Based User Location and Tracking System. In *INFOCOM (2)*, pages 775–784, 2000.

[5] D. Barbará and T. Imieliński. Sleepers and workaholics: caching strategies in mobile environments. In *ACM SIGMOD*, pages 1–12, Minneapolos, Minnesota, 1994.

[6] T. Camp, J. Boleng, and V. Davies. Mobility Models for Ad Hoc Network Simulations. *Wireless Communication & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, 2002.

[7] P. Castro, P. Chiu, T. Kremenek, and R. Muntz. A probabilistic location service for wireless network environments. In *Ubiquitous Computing 2001*, Atlanta, GA, September 2001.

[8] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *World Wide Web*, pages 265–274, 2001.

[9] W. K. Edwards, E. D. Mynatt, K. Petersen, M. Spreitzer, D. B. Terry, and M. Theimer. Designing and Implementing Asynchronous Collaborative Applications with Bayou. In *ACM Symposium on User Interface Software and Technology*, pages 119–128, 1997.

[10] R. Frenkiel, B. Badrinath, J. Borras, and R. Yates. The Infostations Challenge: Balancing cost and ubiquity in delivering wirelss data. *IEEE Personal Communications*, February 2000.

[11] A. Kahol, S. Khurana, S. Gupta, and P. Srimani. A Strategy to Manage Cache Consistency in a Distributed Mobile Wireless Environment. In *The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, 2000.

[12] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, 1991.

[13] Y. Lin and W. Tsai. Location Tracking with Distributed HLRs and Pointer Forwarding. *IEEE Transactions on Vehicular Technology*, (47(1)):58–64, January 1998.

[14] Q. Lu and M. Satyanarayanan. Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions. In *Fifth IEEE HotOS Topics Workshop*, Orcas Island, WA, USA, 1995.

[15] K. Tan, M. J. Franklin, and J. Lui. Bandwidth-Conserving Cache Validation Schemes in Mobile Database System. In *Proceedings of Mobile Data Management Conference*, Hong Kong, China, January 2001.