Flexible On-Device Service Object Replication with Replets

Dong Zhou
DoCoMo USA Labs
181 Metro Dr., STE 300
San Jose, CA 95110, USA
zhou@docomolabs-usa.com

Nayeem Islam
DoCoMo USA Labs
181 Metro Dr., STE 300
San Jose, CA 95110, USA
nayeem@docomolabs-usa.com

Ali Ismael
DoCoMo USA Labs
181 Metro Dr., STE 300
San Jose, CA 95110, USA
Ismael@docomolabs-usa.com

ABSTRACT
An increasingly large amount of such applications employ service objects such as Servlets to generate dynamic and personalized content. Existing caching infrastructures are not well suited for caching such content in mobile environments because of disconnection and weak connection. One possible approach to this problem is to replicate Web-related application logic to client devices. The challenges to this approach are to deal with client devices that exhibit huge divergence in resource availabilities, to support applications that have different data sharing and coherency requirements, and to accommodate the same application under different deployment environments.

The Replet system targets these challenges. It uses client, server and application capability and preference information (CPI) to direct the replication of service objects to client devices: from the selection of a device for replication and populating the device with client-specific data, to choosing an appropriate replica to serve a given request and maintaining the desired state consistency among replicas. The Replet system exploits on-device replication to enable client-, server- and application-specific cost metrics for replica invocation and synchronization. We have implemented a prototype in the context of Servlet-based Web applications. Our experiment and simulation results demonstrate the viability and significant benefits of CPI-driven on-device service object replication.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *Client-server, distributed applications.*

General Terms
Measurement, Performance, Design

Keywords
Service, replication, preference, capability, reconfiguration, synchronization

1. INTRODUCTION
With the growing popularity of wireless Internet and the advance of mobile access technologies, Web applications are increasingly accessible for wireless devices such as cell phones, PDAs and WiFi-enabled laptops. Particularly, dynamic Web page generation techniques, such as Java Servlet and CGI, have become a trend to provide personalized Web pages and context-aware Web services to the wireless devices. However, traditional means of optimizing dynamic Web content generation and delivery are not well suited to the nature of unstable wireless connectivity. When a wireless client loses connection (either voluntarily or involuntarily) or roams into an area with weak connectivity, common optimization techniques, including static page prefetching [3], dynamic content caching [6][1][14][27][7], data replication [8][16][13][17][15][9][11][18], and content adaptation [5][28][22] as well as application offloading, will fail to maintain effective interactive operations (e.g., querying account information, filling survey forms, checking product inventory) as they do not eliminate the need to go through the wireless connection for dynamically generated content.

One solution to enabling disconnected or weakly connected operations is *on-device service object replication*. As wireless devices become more powerful and possess more capacity, it becomes feasible and foreseeable to replicate some application service objects from the server to client devices. A typical dynamic Web application can be divided into three parts: Web-related application logic, back-end application logic, and application data. On-device service object replication targets the Web-related application logic part, as well as application data relevant to that specific device (or user). An example of user-specific application data is a portfolio of a user's mutual fund account. Note that we assume the Web-related application logic performs varying functions from simple Web operations (such as converting underlying data to HTML form) to complex tasks (such as " thinking" in reaction to user moves in an online board game).

On-device service object replication is by no means a new concept. Previous works such as Coda [20], Rover [12], Active Cache [4], Active Names [24], and Client-side Include [19] have, to varying extents, investigated this concept. However, none of such existing research works fully exploited the main distinction between server managed service object replication (which is mainly for availability and reliability) and on-device service object replication: the asymmetry between the server replica and a device replica and the asymmetry among device replicas, caused by the differences in hardware capabilities and user preferences.

More specifically, these existing works do not adjust their replication strategies for,

- **Divergent device capability**, including processor speed, available memory and power, and network bandwidth. A specific service object may be replicable to some devices but not to others. Conversely, a device may be able to replicate some service objects but not others.
- **Different application, user and server needs**, such as data consistency requirements and latency tolerance levels. For example, devices with different degrees of connectivity call for different degrees of data consistency or freshness. A device with WiFi connection can subscribe to live streaming stock quotes, while a device with GPRS connection charged

by number of packets may only need a refresh at the end of the day.

- **Evolving environmental conditions**, such as changing connectivity or changing server load. A device moving between good and bad connection may want to change its choice of caching strategies, and a server under changing load condition may adjust limits on clients' access rates.

Such limitations confine their applicability and the range of applications they support.

In this paper we describe Replet, a flexible system for customizing and dynamically adapting the replication of application service objects to user devices. At the center of the Replet system is a mechanism for the user devices, the application and the server to specify their preferences and expressing runtime capability. The preference and capability information from different entities might be conflicting or overlapping. For instance, the current server load and the behavior of clients can yield different preferences in tolerable latency for service responses. The system gathers preferences and capability information from different entities, combines them, and resolves any conflicts among them.

Preference and capability services provide the foundation for the customizability and adaptability of the replication process of Replets. In particular, the invocation of a Replet involves decision objects using runtime capability information to decide, for each service request, whether to invoke the Web-related application logic replicated on the device, or its counterpart on the server. Such decision objects, called Client-side Invocation Helper (CIH) and Server-side Invocation Helper (SIH), are defined through preferences and thus can come from the device, the server, or the application. A sample CIH is a decision object that chooses the replica with lower estimated response time between invoking device replica and server replica, and a sample SIH is a decision object that always chooses the server replica unless a server load threshold is exceeded.

Likewise, the synchronization of application data in the Replet system involves querying a Synchronization Helper (SH) acquired from preferences to determine when to sync application data replicated on device with the copy residing on the server, and whether to lock server copy to ensure serializability. A sample SH is one that suggests synchronizing data only when no such synchronization has happened yet for that day (or week).

Replet addresses the divergent device capability problem as it uses device capability information and application characteristics to guide the selection of devices for replication. It addresses different application, server and user needs by allowing each of them to specify customized invocation and synchronization strategies. It addresses the evolving environmental condition issue by profiling of runtime capabilities that dynamically guides the invocation and synchronization processes to adapt to environment changes.

We have implemented a prototype of Replet system and have converted several servlet applications into Replet services. The prototype does not disable existing devices without Replet, which can still access Replets as a normal servlet application. For clients with Replet support, our experiments show that Replet can improve system performance by reducing response time and network traffic for client, reducing server load, and enabling disconnected operation.

Our contributions are as follows: First, to the best of our knowledge, Replet is the first system for runtime on-device replication targeting dynamic Web applications. Second, our preference and capability centered approach allows per-request dynamic selection of replica invocation, incorporating client-, application- and server-specific cost metrics. It is the first to introduce dynamic replica selection in on-device service object replication. Third, the same preference and capability-centered approach combined with the data synchronization framework gives Replet system extremely flexible support for a broad spectrum of consistency schemes. Fourth, our prototype system and performance experiments show that, without disabling existing clients that do not have Replet support, the Replet system can improve system performance by reducing response time and network traffic for client, reducing server load, and enabling disconnected operation.

The rest of the paper is organized as follows: Section 2 presents the Replet model, its preference and capability services, and the process of Replet replication. Section 3 describes a prototypical implementation and related experiences. Section 4 evaluates the system with experiments and simulations. We compare Replet system with related work in Section 5. Section 6 concludes the paper.

2. REPLET SYSTEM

In this section, we first present the Replet model, followed by discussion on capability and preference management in the Replet system. We then describe in detail how to use capability and preference information for flexible on-device service replication.

2.1 Replet Model

A *Replet* is a replicable object, and is part of a *Replicable Service* deployed on a server. A Replicable Service consists of one or more Replets and, optionally, other non-replicable objects. Each Replet of a Replicable Service represents one service interface of the Replicable Service. A Replet handles the requests from clients and generates results for these requests. Each Replet can have multiple replicas, one of which is a primary replica (or *Server Replica*) that resides on the server, while others reside on client devices (henceforth *Client Replica*). Intuitively, a Replet is similar as a usual service object such as a Servlet [23], except that it can be copied to client devices and serve requests locally.

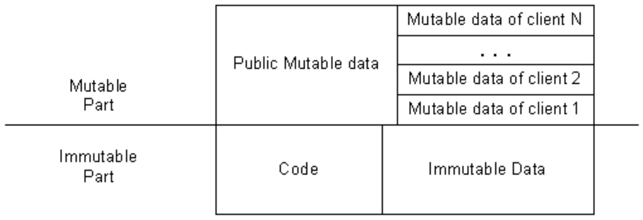

Figure 1: Dissection of a Replet Server Replica

A Replet replica is explicitly divided into code, immutable data and mutable data (Figure 1). The code part includes the class files that define the Web-related application logic, and it is identical for all replicas of the same Replet. However, the mutable and immutable data (which can be a combination of in-memory objects, files and mobile database tables with records and attributes tailored to client's needs) of a specific Client Replica can be different from that of a Server Replica or other Client Replicas. For example, a Client Replica can have rows of a database table filtered out, which are different from another Client Replica. The mutable data of a replica, which can be modified by clients, is further divided into a public fragment and a private fragment. The public fragment is shared by a number of clients, thus is accessible to the Server Replica and the Client Replicas on those clients. The private fragment is specific to a client and is only accessible to the Client Replica and its Server Replica.

Figure 1 depicts a Server Replica. Client Replica on a client device is slightly different in that it only has the private mutable data for that given client.

At a given moment, for a given client, a Server Replica can be in one of following three states (Figure 2):

- *App-Synchronized*: the Server Replica has up-to-date public mutable data but does not have up-to-date private mutable data for the client;
- *Client-Synchronized:* the Server Replica has synchronized copies of both public mutable data and the private mutable data for that given client;
- *Invoked:* the Server Replica was in Client-Synchronized state and has been selected to serve a request from the client, and the invocation is in the process.

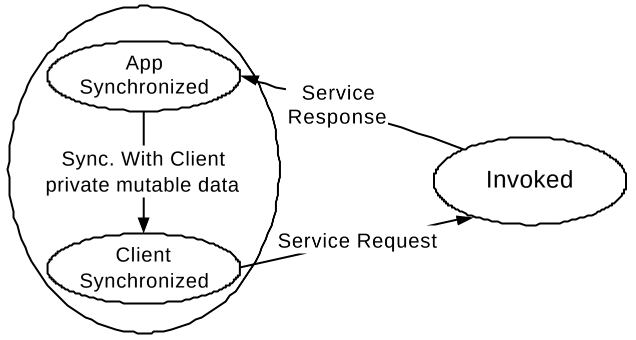

Figure 2: State transition of a Server Replica

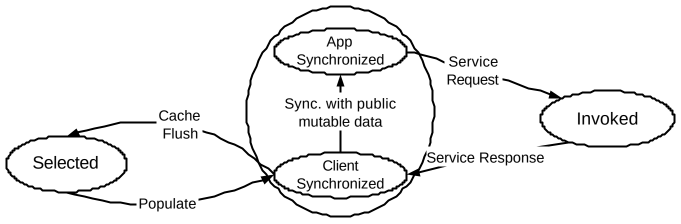

Figure 3: State transition of a Client Replica

Note that a Server Replica can be in Client-Synchronized state for one client, but in App-synchronized for another.

A Client Replica has an additional *Selected* state, meaning that the Replica has not yet been populated with code and data, or the code and data have been removed to allow the replication of other applications (Figure 3). Note that after invocation, a Server Replica transits into App-Synchronized state, while a Client Replica transits into Client-Synchronized state.

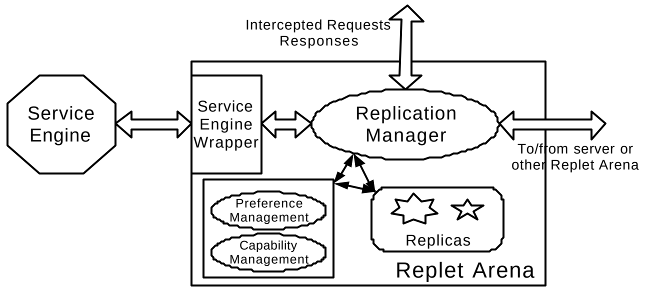

Figure 4: Replet Arena

Replicas of Replets live inside *Replet Arenas* (Figure 4). A Replet Arena is a runtime environment for replicas of different applications. A replica in an Arena can either be the Server Replica of an application or a Client Replica of the application. A Replet Arena consists of following components:

- The *Service Engine Wrapper*
- Preference and Capability Managers
- The Replication Manager

The Service Engine Wrapper implements the *Service Engine Interface* and is the delegate for the external service engine chosen by the Replet Arena. The Service Engine Interface defines methods for deploying and removing a service, and for getting a service container for a particular service request to serve the request. While the Service Engine Interface is identical for all Replet Arenas, different Arenas use different Service Engine Wrappers when they employ different service engines. For example, one Replet Arena running on a desktop computer could use the Service Engine Wrapper for Jakarta Tomcat [2], while another running on a PDA could use the wrapper for Jetty [10] that can be configured to less than 300KB.

Preference and Capability Management provide information used by the Replication Manager to direct the process of replication: from the selection of a client device for replication, and populating the device with code and data, to synchronizing replicas and choosing a synchronized replica for serving a request. We describe Capability and Preference management, and the Replication Manager in detail next.

2.2 Capability and Preference Management

Replet uses *capability* and *preference* information to direct the process of replication. Capability is some system status information gathered or estimated at the runtime, such as available memory, and the response time estimation. Preference information consists of preferences pre-specified by the application server, the user, or the application, such as the required memory and the preferred response time. Comparing preference and capability information, Replet can determine which replica to use or whether to download a replica from the server. In this subsection, we'll describe Preference and Capability Management in detail.

2.2.1 Preference Management

Three participating entities, also called *roles*, take part in preference derivations: the client, the server and the application. But the entities often have conflicting or overlapping preferences. For example, an impatient user wants to get the results in 10 seconds while the application estimates the acceptable response time to be within 15 seconds. Preference Management merges potentially conflicting partial preferences from different roles into a unified global preference.

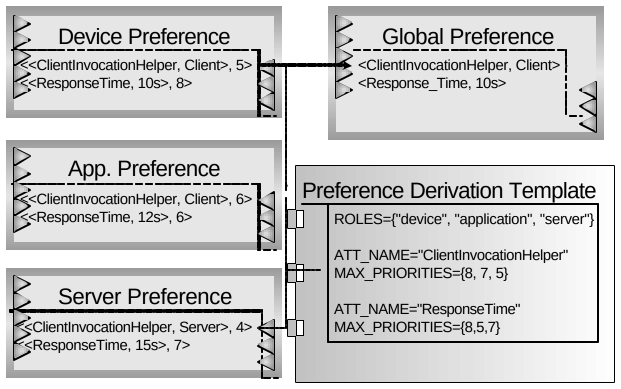
Device Preference
<<ClientInvocationHelper, Client>, 5>
<<ResponseTime, 10s>, 8>
Global Preference
<ClientInvocationHelper, Client>
<Response_Time, 10s>
App. Preference
<<ClientInvocationHelper, Client>, 6>
<<ResponseTime, 12s>, 6>
Preference Derivation Template
ROLES={"device", "application", "server"}
ATT_NAME="ClientInvocationHelper"
MAX_PRIORITIES={8, 7, 5}
ATT_NAME="ResponseTime"
MAX_PRIORITIES={8,5,7}
Server Preference
<<ClientInvocationHelper, Server>, 4>
<<ResponseTime, 15s>, 7>

Figure 5: Examples in Preference Management

Figure 5 illustrates how Preference Management works. Each entity specifies a partial preference that consists of a set of *partial properties*. A *partial property* is a `<property, precedence>` pair, where `property` is a *property*, while `precedence` is a number denoting priority. Each *property* is a `<name, value>` pair, where `name` is a string and `value` can be an arbitrary object. For instance, <<ResponseTime, 10s>, 8> in Device Preference means that the device preferred response time should be within 10 seconds with a precedence level of 8.

In order to derive a global preference, we first validate each partial property with *Preference Derivation Template*. The Preference Derivation Template sets maximum priority levels for each partial property. If a partial property contains a precedence value higher than the maximum level specified in the template, the partial property will be considered invalid. For example, ResponseTime in Application Preference is invalid because its precedence (i.e., 6) exceeds the maximum priority (i.e., 5).

Afterwards, we merge all valid properties into Global Preference. If there are conflicting properties, we only keep those with higher precedence values. For example, there are two valid ResponseTime properties specified in Client Preference and Server Preference, respectively. We keep the one in the Client Preference because its precedence value (i.e., 8) is higher.

2.2.2 Capability Management

Replet Arena's capability service maintains capability and profiling information of the client, the server, and the application. Client capabilities include amount of memory or storage available on the device, its CPU processing power, the availability of just-in-time compiler, down- and up-link bandwidth, and round-trip time to the server. A server's capabilities include its CPU processing power, and its load. Application profiling data includes storage and memory requirement on some typical device types, the range of costs for processing requests on several typical devices, observed responses time for requests, current number of client replicas for the Replet, as well as the application's requirement for the software environment (such as requiring a specific JAR file).

Some of such capabilities or characteristic data (such as installed memory) are static, while others (such as current server load) are dynamic. For dynamic information, the Capability Profiler also provides statistics for such information, and regression tools for fitting and prediction.

Since capability information has distributed sources, Capability (and Preference) Management of client and server side Replet Arenas need to exchange their local information to construct the global capability information. Such exchange will be described in detail in subsection 2.3. Also, Capability Management allows other components of the program to subscribe to individual capability items, so that they can be notified when the value of such capability items change.

2.3 Replication Process

The Replication Manager in the Replet system manages the replication process of the Replets. A Replet's replication process includes four phases: (1) the selection of a particular client for replication (the Selection phase); (2) the populating of the selected device with code and data (the Populating phase); (3) the invocation of a replica (the Invocation phase); and (4) the state synchronization among multiple replicas (the Synchronization phase). An important distinction between Replet and a typical replication system is that the replication process in the Replet system relies heavily on capability and preference information of the client, the server and the application, so that the entire process is highly flexible.

2.3.1 Replication Device Selection

In Selection phase, the Replication Managers of both client- and server-side Replet Arenas collaborate to determine whether the client device should be one of the replication sites of an application deployed on the server.

The client-side Replication Manager intercepts all the service requests that come from the local client and are targeted to the services residing in remote servers. For services that have not been replicated locally, the Replication Manager will check local Preference (which is derived from locally defined Client Partial Preference and default Server and Application Partial Preferences) and to examine if the local client currently allows Replet replication. If it does, the Replication Manager can then attach an optional Probe Flag to the request that is going to be forwarded to the server. The Probe Flag indicates that the client allows Replet replication.

A server that does not support Replet replication ignores the Probe Flag in the request. However, a server that does support Replet replication, upon receiving such flag in the request, checks if the Replet for the requested service allows itself to be replicated. If it does, then the server attaches Server and Application Partial Preferences and capabilities to the usual service response, and sends them back to the client. Also attached to the response is a Client ID that can later be used to distinguish different clients, and to server as a proof of permission for replication from the server. The Client ID expires after an amount of time determined by the Preference.

Back at the client, when the Replication Manager receives a response with server and application CPI and Client ID, it checks the Replet's resource requirement (such as memory and CPU usage) against resources available on the device to determine whether the device should be a replication site for the Replet. In addition, Server and Application Partial Preferences are used to replace default values to complete the client-side derivation of the Preference.

2.3.2 Populating a Replication Device

The Populating phase starts when the client sends the server a usual service request with an attached Download Flag. Note that this Populating phase can start at any time between the end of the

Selection phase and the expiration time of the Client ID. A client can use this flexibility to start the Populating phase after enough resources have been allocated for replication, or when it expects server to be under lower loads.

The server can choose to send requested code and data back to the client immediately (along with usual service response), or temporarily decline the client's populating request, in which case the client can make another populating attempt later. The code and data to be sent can be specified by the Replet itself, and the Client ID can be used to the further customized the data sent to the specific client.

When the Push Endpoint field is present, the server has the option of asynchronously pushing requested code and data to the client at a time deemed proper by the server, rather than having to attach the requested code and data with the service response. This Push Endpoint is also used for push-based data synchronization (see subsection 2.3.4).

2.3.3 *Service Invocation*

At the end of the Populating phase, the Client Replica of the Replet is created and its state is set as Client-Synchronized, i.e., the replica's client-specific mutable data is up-to-date. A Client Replica in Client-Synchronized state is able to serve service requests that only access client private data, without having to synchronizing public data with the server.

During the Invocation phase, the Replication Manager intercepts the service request from the local client, and then consults the Client-side Invocation Helper (CIH) for suggestions on where to serve the request. A suggestion returned by an Invocation Helper (IH) contains two fields: one indicating whether to use the Client Replica or the Server Replica for the request, the other indicating the confidence of such suggestion. Such suggestion will be followed if the confidence level is higher than a threshold defined by the Preference: if the suggestion is to use the Client Replica, then the request is served locally; otherwise, the Replication Manager forwards the request to the server, and relay response from the server back to the client, where both the forwarded request and response may contain synchronization information for client private data.

In the cases where the client can't make decisions by itself (that is, when the confidence of CIH's suggestion is below a threshold defined by the Preference), it also forwards the request to the server, attaching an Adaptive Invocation Flag to the request, indicating that it is up to the server to decide where to serve the request. Upon receiving such a service request, the server-side Replication Manager consults the Server-side Invocation Helper (SIH) for suggestion. If the suggestion from the SIH is to serve the request on the server, then the Server Replica is used. Otherwise, the request is bounced back to the client for handling.

The challenging issue in service invocation is selecting the right replica to invoke. The suggestions are typically made according to a particular cost model. However, cost models often vary with different applications, devices and servers. There is no optimal cost model that can be predetermined.

Replet provides a flexible cost model framework through its Preference Management. All three entities, the device, the server, and the application can specify an IH class that implements a particular cost model (see Figure 5) in their partial preferences. Preference Management will decide which Helper class to use by merging them in Global Preference. An IH has access to the Preference and profiled capabilities, as well as the replica itself, in implementing the chosen cost model. The following list shows some sample cost models we have implemented IHs:

- A cost model based on user perceived response time, where an IH uses profiled capabilities, including previously measured request processing time, previously measured request and response message sizes, and network characteristics, to compare the estimated costs of using the Client Replica or the Server Replica. A device moves between good and bad connection areas can use this model to improve user experience.
- A cost model based on server load, where the SIH bounces Adaptive Invocation requests back to client (along with updated server load estimator) when the server load is higher than a threshold. The corresponding CIH uses server load estimator to predict current server load, and uses Client Replica when predicted server load is higher than a threshold.
- A cost model based on total amount of communication between the server and the client, where an IH uses profiled data including request and response message sizes, CPI exchange overhead, and data synchronization cost to choose a replica for the request to minimize the communication cost.

2.3.4 *Data Synchronization*

Until now we have assumed that a service invocation either does not access the mutable data of a replica, or that it only accesses the portion of mutable data that is private to the client, for which it is trivial to synchronize and maintain the consistency as there is no concurrent accesses on the same data on multiple replicas.

Data synchronization and consistency management is much more complex when an invocation on a Client Replica reads and/or writes public data of a Replet. The Synchronization phase in Replet Replication is separated into two stages by the invocation phase: a Read Stage before the Invocation phase, and a Write Stage after the Invocation phase (see Figure 6):

- Read Stage: On the client side, when the Replication Manager decides to serve a request locally, it consults the client copy of the Synchronization Helper (SH) specified by the Preference to see if there is a need to read the most up-to-date version of the Replet's public data from the Server Replica. On the server side, when such read request is received, the Replication Manager consults the server copy of the SH for the client to determine if it is necessary to apply a write-lock to the Server Replica to prevent concurrent accesses. The lock expires after a time defined by the Preference. Note that since there is one Preference for each client, potentially each client can have a different SH.
- Write Stage: On the client side, after an invocation on the Client Replica that modifies the Replet's public data, the Replication Manager again consults the SH to see if there is a need to propagate the modification to the Server Replica immediately. On the server side, when such modification propagation message is received, the Replication Manager consults the SH of the client to potentially detect update conflicts and resolve such conflicts. The Replication Manager then releases the lock on the Server Replica if the client acquired one during the Read Stage. Finally, the Replication Manager consults SHs of *other* clients to see if the modification needs to be pushed to these clients.

Note that the actuation of the Read and Write Stages depends on whether the invoked method reads or writes public data. If the method does not read or write public data, then the Read or Write Stage is bypassed.

SH is also part of the Preference, and can potentially be defined by the client, the server, or the application. Some sample consistency maintenance schemes that can be implemented using SH include:

- Pessimistic replication, where one-copy serializability is achieved by demanding to refresh from Server Replica before each invocation that reads Replet public data, lock the Server Replica after refresh, and contact Server Replica after invocation for update propagation and lock release.
- Optimistic replication with 1/K synchronization, where the data refreshment and update propagation is carried out once for every K invocations that access public data, and no lock is applied on the Server Replica.
- Optimistic replication with push approach, where the client depends on updates pushed from the server to maintain the freshness of the Replet's public data.

The pessimistic replication scheme can be practical if only a small fraction of the invocations involve accesses to public data, and/or if data synchronization overhead is much less costly than the overhead associated with the transport of request and response messages and the processing of the request on the server. Other sample consistency schemes include one that chooses right update rate depending on client capability and current server load for self-refreshing Web contents, and another enforced by a server under unusually high load to temporarily disable a device from reloading the content until sometime later.

3. IMPLEMENTATION

We have implemented a prototype of the Replet system in Java for Web-based applications that use Servlets as service objects. Figure 7 illustrates the high-level architecture of this prototype. We chose Tomcat Servlet container from the Apache Jakarta Project as service engine for the prototype, and we wrote the Service Engine Wrapper for Tomcat.

Replet Arenas are implemented as standalone processes. For a given application, the server-side Replet Arena (the Arena that hosts the application) functions as a Web server with Servlet (and Replet) support, while the client-side Replet Arena is configured as a HTTP proxy of the client Web browser. Server-side and client-side Replet Arenas communicate using HTTP protocol. Fields of Replet replication protocol messages (such as various flags, CPI, IH suggestions, and synchronization data) are piggybacked with HTTP requests and responses in the form of HTTP header fields.

A Replet extends J2EE HTTPServlet class, and implements the Replet interface. The Replet interface specifies methods that can be used by the Replication Manager to:

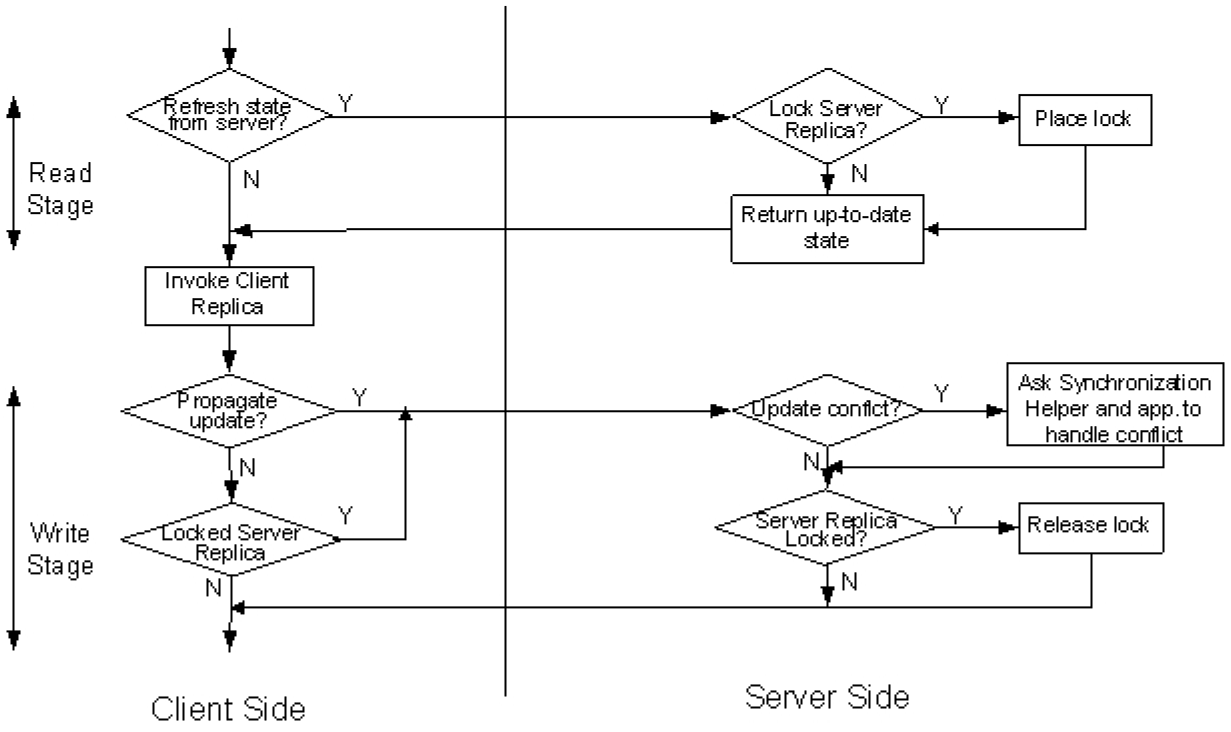

Figure 6: Flow chart for data synchronization

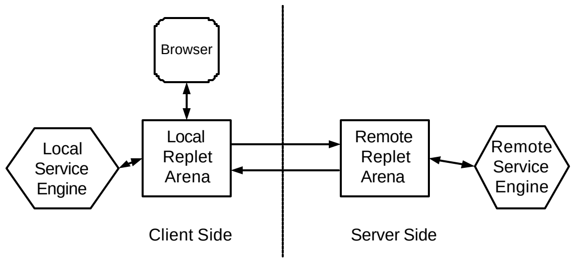

Figure 7: Illustration of prototype implementation

- Get and set client-specific immutable and mutable (both public and private) data: such data can be a combination of in-memory objects, files and (usually filtered) database tables. The default implementation of these methods assumes that there is no immutable data, and that the public mutable data consists of the entire Web archive (WAR) file of the replicable service application except those under WEB-INF/classes and WEB-INF/lib, while the private mutable data consists of the HTTPSession object created for the client.
- Retrieve and apply updates for synchronization: updates can also be represented using in-memory objects, files and tables. The default implementation assumes that the entire Web archive has been updated.

The development of a Replet also involves defining the Application Partial Preference of the Replet, which includes specifying the classes for IHs and SH, as well as their respective initialization arguments. We have ported four originally Servlet-based applications using this prototype: WebMail, which replicates emails to the client device along with the object that handles emails; WebCalendar, which has both private and shared calendar pages; WebChess, where users login to their account to play against computer at different skill levels; and PurchaseApproval, which emulates a sample corporate workflow.

During the course of implementing the prototype and the sample applications, we identified several important issues that we discuss next.

Nonintrusive replication: The replication process should be nonintrusive to both the clients and the server. The most probable source of intrusiveness is in the Populating phase, where the Replet system needs to collect relevant code and data at the server side, send them to the client, and properly install such code and data at the client side. To an already overloaded server, this process may exacerbate the load situation especially when populating is required for many clients in a short period of time; to the client, this process may stall the responsiveness of other requests as receiving the replica and installing it may be costly for some devices. Our solution is to give the server the option of populating a client only when the server thinks it is the right time to do so, and we allow the server to asynchronously push the replica to the client, so that populating can occur between two requests and be less intrusive.

Server load information: A concise representation of dynamic server load information that can be used to closely predict future server load is important in the Replet system, as it can help optimize the performance of the system with light cost. We have experimented with the polynomial regression approach, and our conclusion is that this approach offers similar precision with more compact representation for the server traces that we targeted (see Section 4.3). We also limited the dissemination of server load information so that it is piggybacked with some of the responses to the client.

Session migration: Session migration is necessary to support replication in the middle of a client session, and it is the default form of populating client device with client-specific mutable data. Our session migration implementation relies on cookie-based HTTP sessions. During the migration, the client-side Replet Arena creates a local session cookie and a new session object. The new session object will be populate with contents from the original session on the server session, and the newly created cookie will be returned to the client.

Public data access directives: We use XML-form external directives to denote where the GET and POST handling methods of a Replet reads or writes public data. However, a Replet is typically used for different types of requests. Some of these requests may access public data while others may not. To distinguish such request, an approach similar to J2EE Servlet mapping is used. Each request type is allocated a virtual Replet. While these virtual Replet map to the same real Replet, public data access directives are generated for each individual virtual Replet. For example, in the WebCalendar application, creating an appointment accesses public data while viewing an appointment does not. Although the same Replet handles these two operations, their URI maps to different virtual Replets with different access directives, so that the viewing operation can always be operated locally for a Client Replica.

4. EVALUATION

In this section we evaluate the viability and benefits of on-device service object replication in the Replet system based on our prototype. We describe experiments and their results using some of the sample applications we built, as well as simulations that are based on one of those sample applications.

Table 1: Comparing size cost of replica populating against typical responses (in KB).

	WebChess (KB)	WebCalendar (KB)	Purchase Approval (KB)
Populating	37 - 40	28 – 29.5	73 - 75.5
Typical Response	6.5 – 9.6	4.2 – 6.3	5.5 - 6.7

4.1 Replica Populating Cost

In this experiment, we compare the cost of replica populating against that of typical responses. Table 1: lists number of bytes in a typical HTTP response (including images), and the number of bytes to be shipped for replica populating (which also includes images) for three applications. Considering each populated application can produce multiple responses (this is the case even for Purchase Approval, where a single approval may involve multiple steps, and the user may check approval status for several times), the cost of replica populating is not prohibitively high compared with costs for normal service responses. For example, Figure 8 examines the response time for the computation-intensive WebChess application. The X-axis represents the number of steps of user interaction while the Y-axis represents the response time. The upper two curves show the response time with and without replication, including the populating cost. We can see that the response time drastically drops after replica populating occurs at Step 4. In fact, it becomes very close to the response time during a low server CPU load (shown as the bottom curve). The server used in this experiment is a desktop with 2Ghz P4 processor and 512MB RAM, and client is a ThinkPad T23 laptop connected to the server via IEEE 802.11b wireless network. The load on the server is created by running adjustable compute-intensive tasks in the background.

4.2 Divergence in Request Processing Costs

This experiment uses two applications, WebChess and WebCalendar, and two client devices, an iPAQ 3870 and a ThinkPad T23, to demonstrate that profiled device and application capability is important for on-device replication. Table 2: shows average processing times of the two applications on the two devices. Apparently, while the iPAQ, whose JVM does not have just-in-time compiler, is capable of serving requests for WebCalendar, it is not capable of serving requests for WebChess. This experiment demonstrates that device and application capability information is important in the selection of a device to replicate an application.

Table 2: Request processing time for WebChess and WebCalendar on two client devices. (in second)

	Tablet PC (sec)	IPAQ 3870 (sec)
WebChess	0.8 – 3.2	5.5 - 18
WebCalendar	0.01 – 0.095	0.03 – 0.029

4.3 Server Load Prediction

This experiment demonstrates the effectiveness of using polynomial regression for the coarse-grained prediction of server load using historic information. We arbitrarily selected two weeks' trace of the NASA Web server [21]. To predict the server load on any given day of the second week, we use a cubic function to fit the load of that day of last week, and then use the cubic function to predict the load on that day of the second week. Figure 9 shows two curves: one for real load on the server during the week of July 9-15, 1995, the other for predicted load for the same week based on the actual load information of the previous week. The load prediction curve is actually the combination of 7 curves, each a cubic polynomial for one day. The graph shows that, although polynomial regression over historic data cannot precisely predict future server load, it is sufficiently accurate for client to distinguish lower load periods from high load periods.

4.4 Simulation of Consistency Models

In this subsection, we use simulation to demonstrate the need and benefits of having client-specific consistency models and client-adjustable data freshness requirements. WebCalendar is used as the sample application to derive base costs. We assume that for each client there is a private calendar page and a public page. There are four types of operations, one that reads (but does not write to) the public page (PUB-R), one that reads and writes to the public page (PUB-RW), one that reads private page (PRIV-R), and one that reads and writes to the private page (PRIV-RW). The assumed combined size (including header) of a HTTP request and its response for each type of operation, as well as different distribution of each type of operation for different scenarios, are listed in Table 3:.

We use Poisson distribution for request arrival rate, using C-RATE to denote the rate for requests generated by the client under inspection, while S-RATE for server observed overall rate (requests from all clients except the one under inspection). We fixed C-RATE at 6.67 for all experiments. The real unit of C-RATE does not matter as only the ratio between C-RATE and S-RATE is of interest.

Table 3: Parameters for the simulation

	PUB-R	PUB-RW	PRIV-R	PRIV-RW
Request+Response size	6KB	4.5KB	5KB	4KB
Balanced	35%	15%	25%	25%
Public-Read Biased	80%	5%	10%	5%
Public-Write Biased	50%	40%	5%	5%

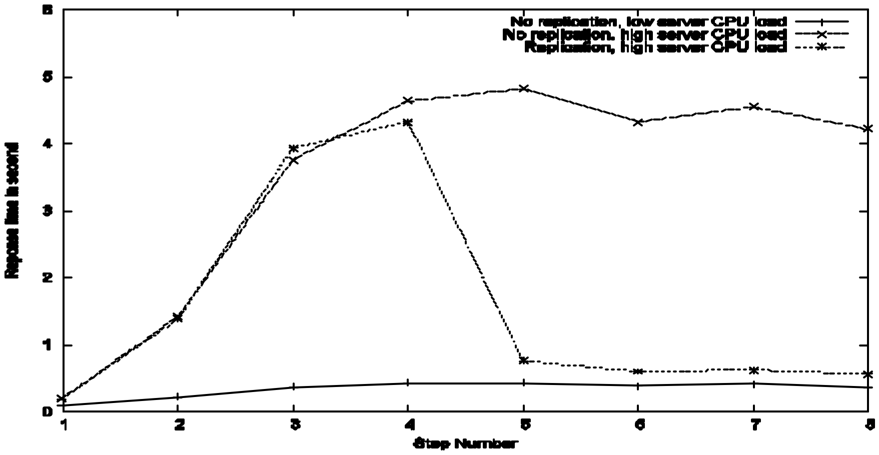

Figure 8: Response time for the WebChess application (on-the-fly replica populating happened at step 4. Note that different steps require different amount of computation)

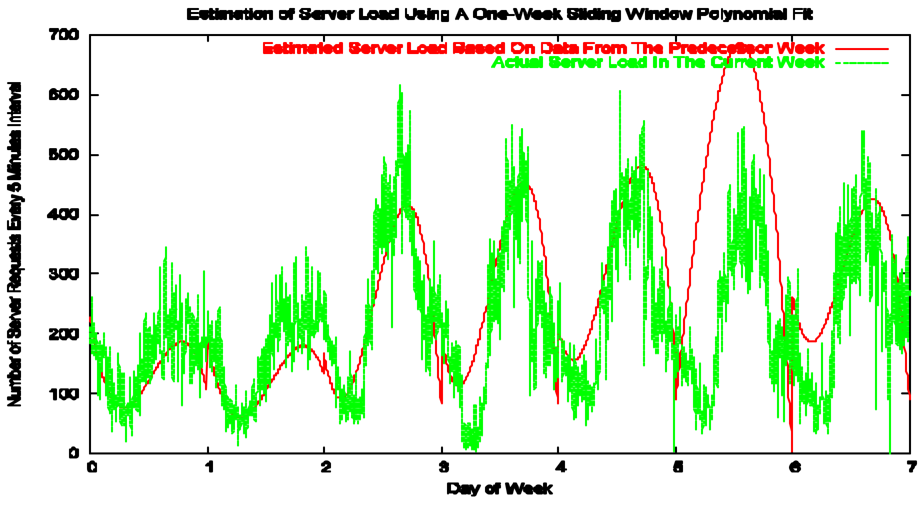

Figure 9: Using Cubic Polynomial Function For Server Load Prediction.

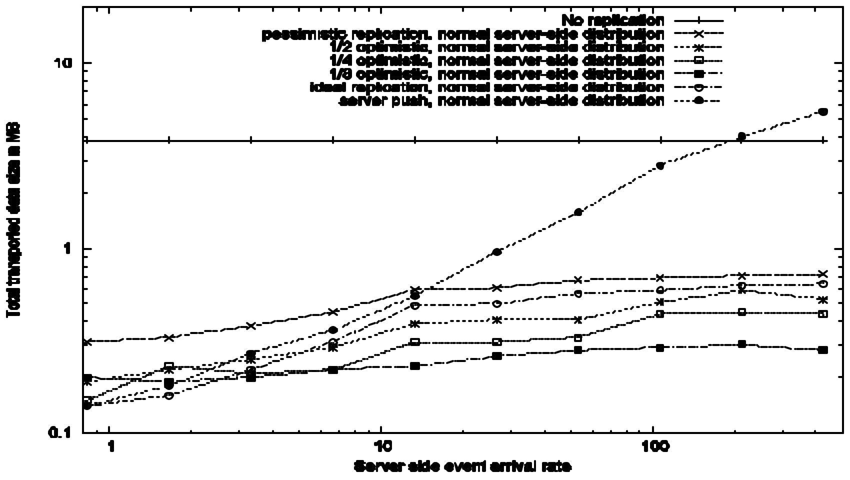

Figure 10: Changing S-RATE under Public-Read Biased distribution.

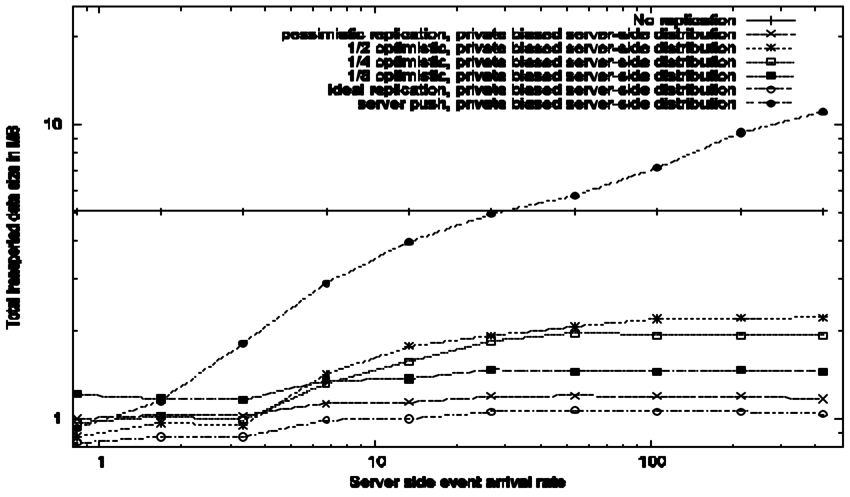

Figure 11: Changing S-RATE under Public-Write Biased distribution.

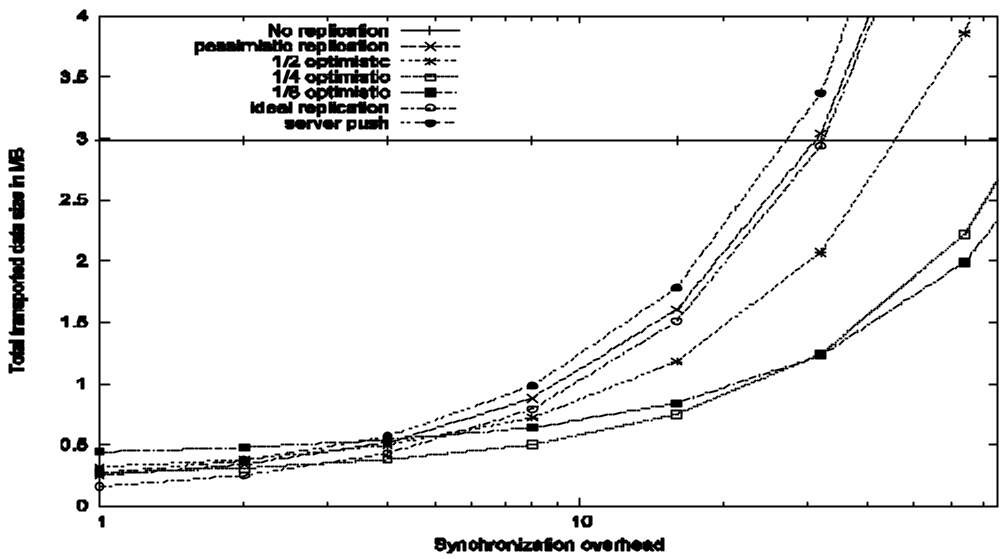

Figure 12: Changing Synchronization Overhead (in KB)

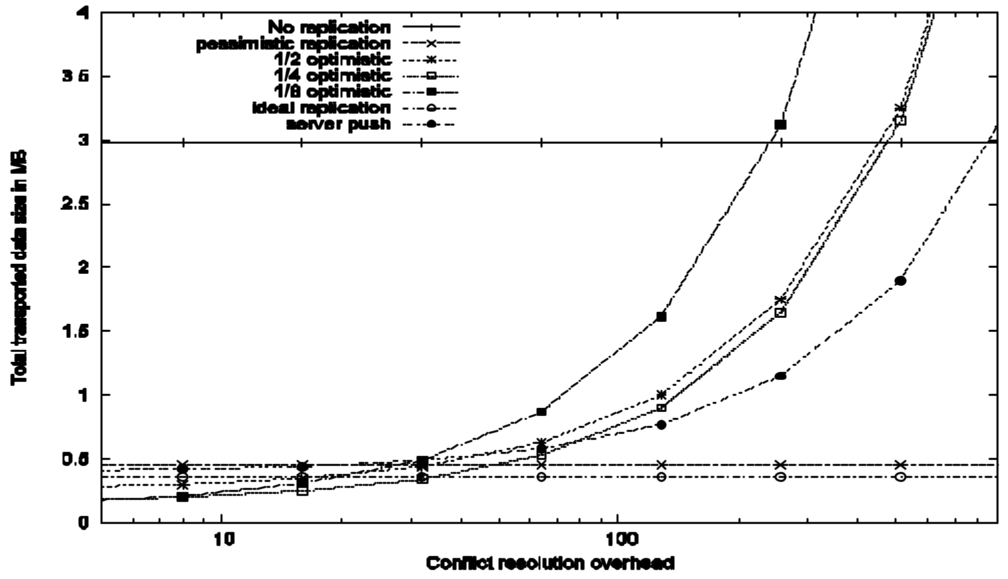

Figure 13: Changing Conflict Resolution Overhead (in KB)

We compare the total amount of traffic for the inspected client under five different schemes: no replication (NR), pessimistic replication (PR), optimistic replication with 1/N synchronization (1/N), ideal replication (Ideal) and, server push (Push). The Ideal scheme assumes that a client knows every past operation of other clients, so that in the Read-Stage, it knows whether it is necessary to refresh state from the server, but in the Write-Stage, it will immediately propagate its updates regardless future operations on shared state. In the Push scheme, the server pushes modification to the clients and clients never pull from the server in Read-Stage. The Push Latency, the time before it reaches the client is made as a linear function of the server arrival rate. We set the default costs for Read-Stage and Write-Stage synchronization to 2KB when such synchronizations result in communication with the server, and we set the default cost for conflict resolution to 32KB. The optimistic scheme is further divided into 1/2, 1/4 and, 1/8 three configurations, representing synchronizing every 2, 4, and 8 public data accesses.

Figure 10 shows the communication cost on the inspected client when the requests to the server are Public Read Biased, while in Figure 11 the requests to the server are Public Write Biased. In both graphs, S-RATE changes from 0.83 to over 100. Both graphs show that the Push approach has higher costs because of the update conflicts caused by the Push Latency. All other approaches that use replication have lower costs than the No Replication approach. Because reading stale information does not incur penalties similar to conflict resolution cost, the 1/N approaches are better than the Ideal approach in Public Read Biased distribution at the cost of reading out-of-sync public mutable data. The Pessimistic approach under Public Write Biased distribution offers performance closest to the

Ideal approach as it avoids update conflicts that are frequent under this scenario.

Figure 12 compares the cost of different schemes when we fix the cost of conflict resolution at 32KB, and vary the cost of synchronization from 1KB to 64KB. S-RATE is set at four times of C-RATE. Figure 13 compares the cost of different schemes when we fix the cost of data synchronization at 2KB, and vary the cost of conflict resolution from 1KB to 512KB. Again, we set S-RATE at four times of C-RATE. In both simulations, we assume distributions of operations are " Balanced". The two graphs show that optimistic approaches have advantages when synchronization cost increases. But they are penalized when conflict resolution cost gets higher. These simulations confirm the value of flexibility in supporting different synchronization schemes and consistency models for different applications and application deployment scales.

4.5 Analysis of CPI Overhead

Overhead of CPI is mainly from the Selection phase and the Populating phase: the server will send server and application CPI to the client if the server agrees to let the client to replicate the service, and the client need to send its CPI to the server when it requests to start populating. Such overhead is limited since:

- They are piggybacked with HTTP protocol messages.
- They occur once during the lifetime of a Client Replica.

CPI data items are compact, with various Helpers being only exceptions. However, we don't transport Helper object, rather, we only transfer Helper class names and initialization arguments. We assume Helper classes are retrieved through trusted code repositories and can be cached on client devices and on server.

5. RELATED WORK

Previous work on server-managed replication is abundant in the literature [8], [16], [13], [17], [15], [9], [11], and [18]. In such systems, decisions involved in the replication process are typically solely made by the server and client devices are not candidates of replication sites. Hence server-managed replication does not support disconnected environment. By contrast, the Replet system employs on-device service object replication that enables disconnected operations.

Coda [20], CSI [19], and Rover [12] are replication systems that support on-device replication for mobile and disconnected computing. These systems, however, do not specifically target dynamic customization of replication. On the contrary, Replet incorporates client capability and preference into the replication process, enabling client-specific invocation and synchronization cost metrics.

There has also been a lot of research on optimizing dynamic and personalized Web content, including dynamic data caching [1][1][14][27][7] and content adaptation [5][28][22]. However, their approaches do not support disconnected computing because they do not cache, replicate or adapt service objects at the client side. In comparison, Replet replicate service objects to the client device, enabling disconnected operations.

Active Cache [4] is closely related to our system in that it associates a server-supplier code called Cache Applet with each URL. When caching a document, a proxy also fetches the corresponding cache applet, which can be invoked when a user request is received. However, targeting caching proxies rather than end-user devices, Active Cache cannot adapt the caching/replication process to different client devices because it does not exploit client information. In addition, it lacks the flexibility in synchronization among Cache Applets of the same document. Compared with Active Cache, our approach provides support for client divergence in both replication and synchronization.

Our work is also different from the general Java Applet framework. A Replet is intended to be executed on the server. It is only installed and invoked on the device for performance optimization. For clients of the same Replet, some of them may opt to install and run in on-device, while others don't. Even for the same client, the decision of which replica to invoke can be made at runtime on per-request basis. In addition, our system provides build-in support for synchronization between a client and its server, which is not available in the Applet system.

Composite *Capabilities/Preferences Profile (CC/PP)* is a framework for describing and managing a user agent's capabilities and the user's preferences for optimizing content processing and display [25][26]. While CC/PP is widely adopted and offers sophisticated implementations, it only focuses on client CPI and does not address server and application CPI, and does not address the runtime merging and reconfiguration of it.

6. CONCLUSION, LIMITATIONS, AND FUTURE WORK

In this paper we have presented Replet, a system that uses client, server and application capability and preference information to achieve flexible on-device replication of service objects, and to incorporate client-specific cost metrics for replica invocation and synchronization. We have implemented a prototype of the Replet system in the context of Servlet-based Web applications. Our experiments and simulations demonstrate the viability and significant potential benefits of employing on-device service object replication with Replets. Such potential benefits include reducing response time and network traffic for client, reducing server load, and enabling disconnected operation.

Our current system does not provide direct support for update conflict detection and resolution, and it does not address security issues involved in on-device service replication. Some of these issues will be the focus of our future research. And for other issues, we will look for available solutions to apply to our system

7. ACKNOWLEDGMENTS

We thank Sajithkumar Kizhakkiniyil and Dr. Wei Han for their help during different phases of the project. We also thank Drs. Lawrence Brakmo, Manuel Roman for reviewing the paper and providing helpful suggestions. We greatly appreciate insightful comments from anonymous reviewers of the conference, as well as those from Drs. Minoru Etoh and Calton Pu.

8. REFERENCES

[1] Akamai Inc. Turbo-Charging Dynamic Web Sites with Akamai EdgeSuite. White paper. 2002.

[2] Apache Software Foundation. The Apache Jakarta Project. http://jakarta.apache.org/tomcat/

[3] AvantGo. http://AvantGo.com

[4] Cao, P., Zhang, J., and Beach, K. Active Cache: Caching Dynamic Contents on the Web. In Proceedings of Middleware '98, Sep. 1998.

[5] Chen, Y., Ma, W.-Y., Zhang, H.-J. Detecting web page structure for adaptive viewing on small form factor devices. WWW 2003: 225-233.

[6] Datta, A., Dutta, K., Thomas, H., VanderMeer, D., Suresha, and Ramamritham, K. Proxy-based acceleration of dynamically generated content on the world wide web: an approach and implementation. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, June 2002.

[7] Gao, L., Dahlin, M., Nayate, A., Zheng, J., and Iyengar, A. Application specific data replication for edge services. In Proceedings of the Twelfth International World Wide Web Conference (WWW), May 2003.

[8] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, D., Sidebotham, R., and West, W. Scale and performance in a distributed file system. ACM Trans. on Comp. Sys. (TOCS), 6(1), 1988.

[9] Huang, Y. and Wolfson, O. A competitive dynamic data replication algorithm. In Proceedings of Ninth International Conference on Data Engineering (ICDE), April 1993.

[10] Jetty:// Web Server & Servlet Container. http://jetty.mortbay.com/jetty/index.html

[11] Johnson, G., and Singh, A. Stable and fault-tolerant object allocation. In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC), July 2000.

[12] Joseph, A., deLespinasse, A., Tauber, J., Gifford, D., and Kaashoek, M. Rover: A Toolkit for Mobile Information Access. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP), Dec. 1995.

[13] Kim, M., Cox, L. P., and Noble, B. D. Safety, visibility, and performance in a wide-area file system. In Proceedings of USENIX Conf. on File and Storage Sys. (FAST), Jan. 2002.

[14] Li, W.-S., Po, O., Hsiung, W.-P., Candan, K. S., and Agrawal, D. Engineering and hosting adaptive freshness-sensitive web applications on data centers. In Proceedings of the Twelfth International World Wide Web Conference (WWW), May 2003.

[15] Liscov, B., Adya, A., Castro, M., Day, M., Ghemawat, S., Gruber, R., Maheshwari, U., Myers, A. C., and Shrira, L. Safe and efficient sharing of persistent objects in Thor. In Proc. of the ACM SIGMOD International Conference on Management of Data, June 1996.

[16] Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Demers, A. J. Flexible Update Propagation for Weakly Consistent Replication. In Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16), Oct., 1997.

[17] Pu, C., and Leff, L. Replica Control in Distributed System: An Asynchronous Approach, In Proc. of the ACM SIGMOD International Conference on Management of Data, May 1991.

[18] Rabinovich, M., Rabinovich, I., Rajaraman, R., and Aggarwal, A. A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service. In Proceedings of 19th ICDCS, June 1999.

[19] Rabinovich, M., Xiao, Z., Douglis, F., and Kalmanek, C. Moving Edge-Side Includes to the Real Edge - the Clients USENIX Symposium on Internet Technologies and Systems. 2003.

[20] Satyanarayanan, M. The evolution of Coda. ACM Trans. on Comp. Sys. (TOCS), 20(2), May 2002.

[21] Server trace for NASA Kennedy Space Center WWW server. http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

[22] Steinberg, J., and Pasquale, J.. A Web Middleware Architecture for Dynamic Customization of Content for Wireless Clients. WWW 2002: 639-650.

[23] Sun Microsystems, Inc. Java Servlet Technology. http://java.sun.com/products/servlet/

[24] Vahdat, A., Anderson, T., and Dahlin, M. Active Naming: Programmable Location and Transport of Wide-area Resources. In Proceedings of 1999 USENIX Symposium on Internet Technology and Systems (USITS), Oct. 1999.

[25] W3C. Composite Capability/Preference Profiles (CC/PP): A user side framework for content negotiation. W3C Note. July 1999.

[26] W3C. Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies. W3C Working Draft. March 2003.

[27] Yin, J., Alvisi, L., Dahlin, M., and Iyengar, A. Engineering server-driven consistency for large scale dynamic Web services. In Proceedings of the Tenth International World Wide Web Conference (WWW), May 2001.

[28] Yuan, C., Chen, Y., and Zhang, Z.. Evaluation of edge caching/offloading for dynamic content delivery. WWW 2003: 461-471.