RC25350 (WAT1301-080) January 29, 2013
Computer Science

IBM Research Report

Untether: Middleware Components to Support Intermittently Connected Web-Applications

Avraham Leff, James T. Rayfield, Ravi Konuru
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

Raj Balasubramanian
IBM Software Group

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Untether: Middleware Components to Support Intermittently Connected Web-Applications

Avraham Leff, James T. Rayfield, Ravi Konuru
IBM
T.J. Watson Research Center
Yorktown Heights, NY, USA
{*avraham, jtray, rkonuru*}*@us.ibm.com*

Raj Balasubramanian
IBM
raj_balasubramanian@us.ibm.com

***Abstract*—We examine the lifecycle requirements of *intermittently connected web-applications* (ICWAS) and investigate whether such applications can be developed as an "always connected" web-application combined with middleware that address ICWA requirements. We show that this is difficult to do because ICWAS require application-specific logic that is not easily combined with a middleware API. We therefore propose the use of "middleware components" in the areas of data-provisioning and change-set propagation. Combined with application-specific logic, these components make it easier to develop an ICWA by reducing the amount of required developer code. We show how our prototype UNTETHER system implements these components and reduces the burden on the application developer.**

***Keywords*-mobile application; intermittently connected application; middleware**

I. INTRODUCTION

We define an intermittently-connected web-application (ICWA) as a web-application that reads and modifies non-trivial amounts of data and remains functional even when the application is not connected to the server. Although the technological underpinnings for ICWA construction have existed since the 1990s, recent standards work has finally made the construction of ICWAS feasible. HTML 5 contains several features, most notably the Cache Manifest and "Offline Application Caching APIs" [1], that allow developers to specify which parts of an application should be cached for offline use. The W3C IndexedDB draft standard [2] defines a NoSQL database with query APIs that can locate records either by key or by an index. Developers can now build an ICWA, knowing that it can be deployed directly into major browsers without the need for additional plugins.

An ICWA can be viewed as an extreme case of a "web 2.0" application in the way it implements page transformations in response to user-interactions with the GUI. Rather than requesting that the server build the next page, the client itself builds the next page using cached data and business logic [3], [4]. Similarly, REST techniques [5] are used to provision data onto the device and to copy device-resident data to the server. However, because an ICWA cannot rely on being able to connect to the server, it differs from a connected web 2.0 application (CWA) in the following important ways:

- *provisioning*: an ICWA's data-set is not guaranteed to be resident on the device when needed. Typically, when a CWA needs to populate a grid with, for example, a list of "work items", it issues a query to the server. Because the server can transactionally query the master database, the application receives a correct – and complete – result set. In contrast, builders of an ICWA must calculate, in advance, the data-set required by the application and provision the device before it disconnects from the server. Depending on an application's complexity and how long the device remains disconnected, this can be an error-prone process. (Typically, the data-set that can potentially be accessed by an application is too large to be copied in its entirety to the a device.)
- *change-set propagation*: data are created, modified, and deleted as an application executes. Typically, a CWA propagates its change-sets by issuing a web-request to the server that controls the *shared* master database [6]. Because the server has an up-to-date view of the database, it can transactionally commit the change-set in a way that guarantees that the changes are "valid" and do not violate the application's semantics. Account balances, for example, will not go below pre-specified limits; workers will not be assigned the same work-order. In contrast, an ICWA's change-sets are created while the device database becomes increasingly stale since the disconnected device is not aware of changes made by concurrent users to the master database. The database-locking mechanisms used that are typically used by a CWA [7] are not feasible for a ICWA because these locks must then be held throughout the disconnection period, causing unacceptable performance. Moreover, most application data-sets are so dynamic that it is difficult to even predict what data needs to be locked before the device disconnects from the server.

In this paper, we investigate the implications of these ICWA-specific issues on the challenge of building middle-

ware to support the ICWA life-cycle. Section II discusses the provisioning and change-set propagation issues in more detail. In contrast to approaches in which the middleware provides no persistence and only acts as a pass-through for change-set propagation and provisioning [8], we have constructed middleware that supports these activities. However, we also suggest that traditional middleware approaches that cleanly separate applications from middleware may not be feasible; and show that it is instead possible to provide useful "middleware components" that simplify ICWA development. In Section III, we present UNTETHER, our prototype implementation of these middleware components in the context of a sample ICWA. We summarize our findings in Section IV

II. ICWA MIDDLEWARE

Ideally, the role of middleware is to enable applications to be coded as an "application-specific" portion that is combined with separable – application-independent – middleware portions. Application-developers supply the business logic and invoke middleware APIs such as JDBC ® [9] or JNDI [10] as needed. Middleware providers focus on refining the APIs and on providing a robust implementation. In this Section we examine the difficulties inherent in the construction of "ICWA middleware", defined as "middleware that can be combined with an CWA to construct an ICWA". We'll ground this discussion in the context of a quickly-sketched sample ICWA.

Consider a PURCHASE ORDER ICWA that agents use to create orders even when disconnected from the server. A PURCHASE ORDER consists of N "line items", each of which contains catalog information such as product descriptions and price, along with agent-specified information such as quantity. A PURCHASE ORDER is also associated with workflow-state (e.g., "pending"); the client receiving the line items; and the department to which the PURCHASE ORDER will be budgeted. Assuming that the entire catalog will not fit on the device, ICWA developers have to decide which subsets of which product categories will be provisioned. The ICWA can then be written to search the device database for products to display in the GUI, and to store the created PURCHASE ORDER on the device. Finally, when the device can connect to the server, the ICWA must propagate the change-set: in this case, a decremented department budget and the new PURCHASE ORDER. This can be done through service APIs: e.g., a CREATE_PO command, followed by a sequence of SET_LINE_ITEM commands, followed by a CHANGE_STATE command to put the PURCHASE ORDER in the "pending" state. Service APIs enable the application to change master-database state, to drive server-side workflow, and similar actions. After change-set propagation is completed, the server must update the device database to make it consistent with (a subset) of the current master database. This process must incorporate any concurrent activity (e.g., another PURCHASE ORDER created by a different agent that modified the same department's budget) that occurred while the device was disconnected.

This generic and abbreviated description of ICWA development is shown in the left-portion of Figure 1. The core application – the CWA version of an application – consists of three function blocks: a client-resident portion (device database, GUI, and business logic); a server-resident portion (service APIs and implementation); and a master-database (usually on a separate architectural tier) used by the implementation of the service APIs. An ICWA includes, in addition, the function blocks whose role we explain in Sections II-A and II-B. Middleware plays a very important role in the CWA function blocks: e.g., servlet APIs to manage client-server HTTP interactions; JDBC APIs for the server's interactions with the master database; and jQuery to access and modify the browser's DOM. Such middleware can be used in both CWA and ICWA development. We now consider whether ICWA-middleware can play a similar role in the ICWA-specific function blocks show in Figure 1 that handle provisioning and change-set propagation. By decoupling the middleware that enforces reliability and optimization from the service API that enforces business logic during updates/writes to the database, the architecture can scale to support the reconnection of many clients to the server, issuing replication requests, and synchronization their work *via* the service API.

A. Provisioning

Provisioning a given ICWA means storing data required by the application in the device's database. This data-set can be obvious: for example, developers may "know" that an ICWA requires "all information about customers assigned to a given agent". Even when developers can't initially specify the required data-set, an iterative approach may work: e.g., when beta testers report that the ICWA didn't behave correctly, this may point to a need to expand or refine the provisioning. Provisioning is more difficult when an ICWA performs sub-optimally because it doesn't have the correct data-set – but doesn't completely break in an obvious manner. For example, developers may provision a device with catalog information for a PURCHASE ORDER ICWA, but omit data for certain popular items. The ICWA search function will successfully display catalog information, and it may be hard to detect that the device was under-provisioned. Even trivial provisioning tasks can suffer from subtle bugs. Consider the case where the device is provisioned with information about customers assigned to a given agent. The back-end database schema allows queries of the form SELECT * FROM CUSTOMERS WHERE AGENT = "JOHNSMITH". However, the business uses workflow rules that allow one agent to substitute for another when the latter is absent. This workflow is achieved by (temporarily) granting "janedoe" database privileges (perhaps at the business logic level) to

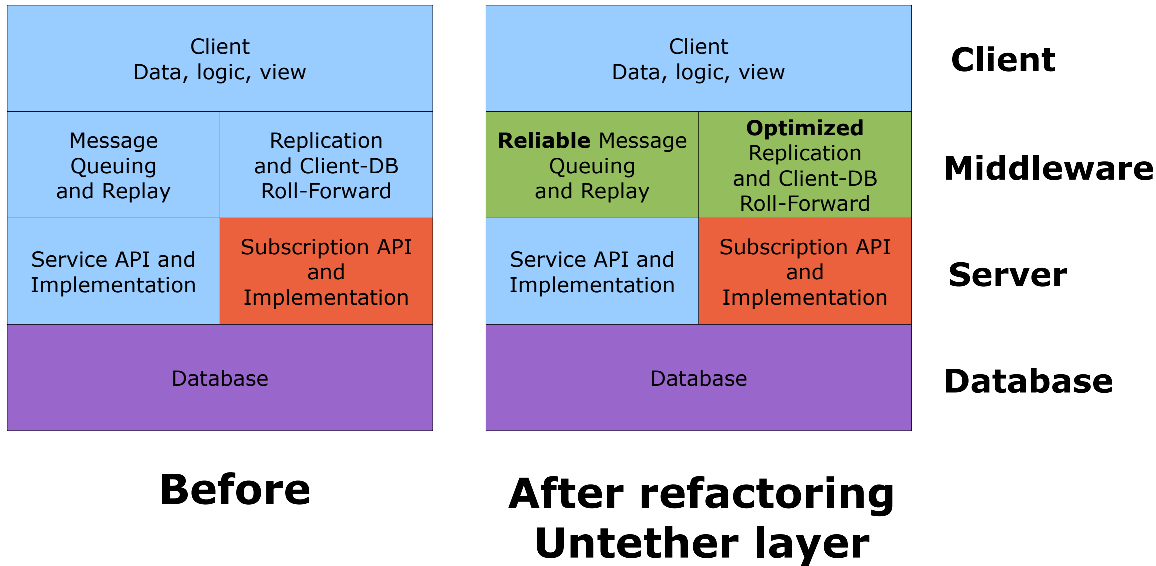

Figure 1. Using UNTETHER To Simplify Develelopment of ICWAs

"johnsmith" data. In the case of a CWA, this approach works seamlessly when "johnsmith" is on vacation. When a request from on of "johnsmith's" customers is fielded to "janedoe", the CWA executing on "janedoe's" device issues a dynamic query to the server which sends back the data she needs. In the case of an ICWA, if developers are unaware of how the workflow interacts with the database schema, and naively provision "janedoe's" device based only on the database query, the provisioning will fail.

The provisioning task relates to the "subscription" and "replication" function blocks in the left-portion of Figure 1. Subscription defines what data are required by a given ICWA; replication copies this data to the device and ensures that the device-database reflects the desired state relative to the master-database and the subscription data-sets. From the perspective of writing middleware, the key point is that whatever its difficulty, provisioning is *application-specific*. It may require relatively little knowledge of the application or a lot, but the task cannot be factored out of the application and delegated to generic middleware. In other words, we assume that the data-set to be provisioned for a given ICWA is specified through a subscription API so that we know which data are relevant to the application and which are not. Our UNTETHER prototype therefore leaves the "subscription" function block unchanged in the right-portion of Figure 1. Instead, the UNTETHER prototype focuses on providing an *optimized replication* module that is used in conjunction with application-specific provisioning. Optimized replication addresses the fact that while the disconnected device has modified its version of the database, concurrent activity (may have) also modified the master database. Usually, the performance of the naive approach that erases the client database and then does a fresh provisioning from the master database is unacceptable. The problem is how to efficiently determine what data in the master database have been modified since a given device last replicated.

Broadly speaking, *timestamp* and *log-based* approaches can be used to solve this problem. In the timestamp approach, every datum (or object) instantiated by a service API is associated with a timestamp. The service APIs are augmented with a GET_ALL_X_SINCE_TIME API to which the client supplies the time that replication was last performed. Because all data are associated with a timestamp, the server can efficiently implement this API to return all

relevant data to the client. (Deleted data is also marked with a timestamp.) In the log approach, the implementation of the service APIs create a log of modified data, and this log drives the creation of a set of "{primary key, timestamp}" records in a new "shadow" database that is separate from the master database. Using the shadow database, the server can efficiently implement the client query to return all data modified since a given time.

Although conceptually similar, the log approach has certain practical advantages over the timestamp approach in the usual case where existing database schema do not already associate a timestamp field with each record. In that case, the log approach is superior because it decouples the infrastructure needed to support an ICWA from existing infrastructure. There is no need to augment existing schema with a timestamp, nor is there a need to modify the existing UPDATE APIs. As discussed in Section III, databases such as DB2 ® and ORACLE® already expose a modification log, making it much easier to automate a log-based implementation. In addition, the shadow database serves as a cache that reduces the load on the master database since it assumes responsibility for the GET_ALL_X_SINCE_TIME queries required by an ICWA.

B. Change-Set Propagation

Two approaches are typically used to propagate the work done by a disconnected device [11]. Here we term them *data-based* and *message-based* propagation.

- Data-based: All data that were modified on the disconnected client are transmitted to the server which, in turn, transactionally commits this state "as is" to the master database. Because the corresponding master-database records cannot typically be locked through the disconnection period, some form of optimistic concurrency control (OCC) [7] is used when committing the records. In the example used in Section II, the client-side database records that were created during PURCHASE ORDER creation are transmitted to the server. These might be represented as a set of line-items, each augmented by a *quantity* column, as well as a PURCHASE ORDER workflow "state" column denoting that the new PURCHASE ORDER must be approved. In addition, the department record whose *budget* column has been decremented, is transmitted to the server.
- Message-based: Instead of transmitting the "raw" database records that have been created, modified, or deleted by the ICWA, the client transmits a sequence of messages that correspond to invocations of the service APIs that are defined for the application. In our example, as the user creates a PURCHASE ORDER, a CREATE_PO message, followed by SET_LINE_ITEM messages, followed by a CHANGE_STATE message are created and stored in the device database. When the device connects to the server, these messages are "replayed", driving the service APIs that create a PURCHASE ORDER, sets its workflow state, and debit the department budget. The ICWA uses the same (typically, lock-based) concurrency control mechanisms that are used by CWA invocations of the service API.

At first glance, the data-based approach to change-set propagation is attractive because its implementation can easily be refactored into middleware. At least for applications whose data are stored in relational databases, the algorithm to represent row and column state as JSON (for example) is straightforward, and is independent of a specific application. Similarly, the process of transforming this state into corresponding JDBC commands that are invoked by the server against the master database is application-independent. However, the data-based approach suffers from several disadvantages that preclude its use in a broad set of application types.

First, note that the data-based approach relies on OCC to provide transactional consistency as the server copies the device's database records to the master-database. That is, the process of committing the device's records must be aborted if the middleware detects that someone has modified a given record in the period between replication and change-set propagation. OCC will have no problem with the *new* PURCHASE ORDER records created by the ICWA, but (even in our very simple example), will abort PURCHASE ORDER creation if some other agent has concurrently decremented the same department budget that is involved with this device's PURCHASE ORDER. The middleware will detect that concurrent work has modified the same database record, and will therefore abort the second transaction. Critically, although it is easy to add business logic to commit the transaction so long as the department budget can pay for both PURCHASE ORDER creations, that is precisely the sort of application-specific logic that is difficult to capture in generic middleware. The message-based approach will have fewer false conflicts than the data-based approach because implementations of the service API contain exactly this sort of application-specific business logic. As long as data operations are commutative, the *exact* values of the data modified by an application should be irrelevant to a service API.

Second, the data-based approach is more fragile than the message-based approach because it imposes no structure on the set of records that are transmitted to the server. For example, if the user created N PURCHASE ORDERs, no demarcation exists between the records associated with one PURCHASE ORDER and the records of another PURCHASE ORDER. The records are transmitted to the server *en masse*, and if any errors occur, the server will have no alternative to aborting *all* of the work done on that device. In contrast, because the message-based approach is based on an application-specific service API, it is relatively easy to

determine which portions of work can be propagated to the back-end database independently of errors detected in other portions of work.

Third, the data-based approach cannot verify read dependencies. Whereas the previous values of changed-data can be verified for consistency at the time the updates are written (OCC), it is not possible to check for the values of read-only data being unchanged at commit time.

Finally, the most important advantage of the message-based approach is that enterprises typically do not permit even a CWA to directly access the back-end database in the manner required by the message-based approach [12]. Enterprise applications are characterized by the fact that business logic is responsible for enforcing system security and integrity. Therefore, those portions of the business logic must be validated by the enterprise before they are allowed to read and write the master versions of application data. Although disconnected operation implies that ICWAs must include substantial amounts of business logic on the client, enterprises typically will not allow client-side execution to substitute for repeated execution of this business logic on the server. Because business logic is responsible for maintaining an application's consistency, integrity, and security constraints, client-side code will not be sufficiently trusted to directly update master-database state. Instead, clients invoke stored procedures [13] which can be thought of as black-boxes of business logic. This has advantages similar to the SOA approach [14] in which enterprises achieve service flexibility by decoupling service API from direct manipulation of the database. Thus, assuming that a service API already exists for a given application, with the message-based approach, ICWA developers use *exactly* the same (already refined and debugged) API already in use by the enterprise's CWA developers.

III. UNTETHER PROTOTYPE

We have built UNTETHER, a prototype end-to-end infrastructure supporting ICWAs through a life-cycle of initial deployment followed by repeated replication, application execution, and change-set propagation events. As shown in Figure 2, UNTETHER consists of four tiers. The master version of an application's data resides in the back-end database (UNTETHER uses DB2). No client-side application is permitted direct access to the back-end database. Instead, client application access is mediated by the back-end server (UNTETHER uses Java™ servlets running on the Jetty webserver v7.44) using an application-specific service API.

Data are propagated from the back-end database to the mid-tier shadow database (UNTETHER uses MongoDB®, a NoSQL database [15]) in a process described in Section III-A. ICWA clients provision data onto their devices through web-requests to the mid-tier server (UNTETHER uses Node.js™ [16]). ICWA clients also do change-set propagation through web-requests to the mid-tier server in a process described in Section III-B.

Provisioned data are stored in the device database (UNTETHER uses IndexedDB [2]). Once provisioning is completed, the PURCHASE ORDER application (written with HTML, CSS, and JavaScript™) can execute without further interaction with the server. Specifically, new PURCHASE ORDERs can be created by querying the *Users*, and *Catalog* object-stores, and storing the new instances in the *PurchaseOrder* object-store. Previously created PURCHASE ORDERs can be retrieved from the *PurchaseOrder* object-store, approved or rejected, and the modified instance updated in the *PurchaseOrder* object-store.

The UNTETHER middleware enables ICWAs to be transformed from the set of function blocks depicted in the left portion of Figure 1 to the set of function blocks in the right portion of the Figure. The core of the application (both on the client and the server) are unchanged. However, by using UNTETHER:

1) Developers can rely on efficient propagation of the data required by the device from the back-end database to the shadow database. Although developers must still specify *what* data are required for a given ICWA, UNTETHER ensures that the most recent data are available and that the device database is moved from its initial state to the current (subset of the) master database.
2) Developers can now queue application-specific commands persistently on the device, and replay them reliably when connected to the server. This allows change-sets to be propagated using using the message-based approach.

We now give more detail about how UNTETHER provides this function.

A. UNTETHER *Provisioning*

As mentioned earlier, we implemented the mid-tier service using Node.js and MongoDB, and the back-end in Java servlets, with the database in DB2. At application start time, the mid-tier server sends a request to the back-end server requesting the initial contents of the database. The back-end server serializes the DB2 tables into JSON, and the mid-term server reads the data and initializes MongoDB.

At this time, the DB2 CAPTURE facility is started [17]. CAPTURE reads the database log, scanning for data written to or deleted from specified tables, and writes the changes to the corresponding "change data" (CD) table. The CD tables generally have the same columns as the original tables, and have additional columns specifying the commit sequence number of the change in the log, and an "operation" column which is either I (for inserted data records), U (for updated data records), or D (for deleted data records). They may optionally have the "before image" of the data, although we do not use this feature in UNTETHER.

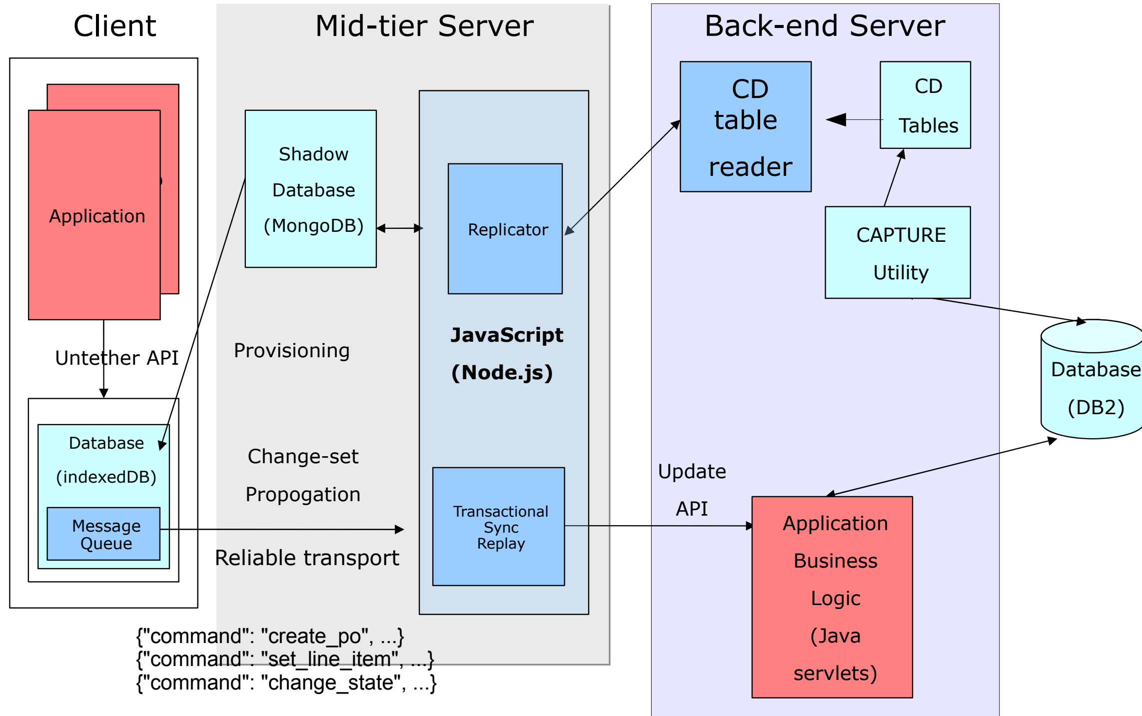

Figure 2. UNTETHER Implementation Block Diagram

For DB2 replication to be enabled, an APPLY program is also required. However, a servlet takes the place of the APPLY program here. The client periodically sends a request to the mid-tier server for any changes that have been made to the database. Then, the mid-tier server sends a request to the back-end server for any changes that have been made to DB2. At that time, the back-end server reads the IBMSNAP_UOW table to find any transactions that have committed to the CD tables. Then, the CD table contents are serialized to JSON and sent to the mid-tier server. Finally, the IBMSNAP_PRUNE_SET is updated with the current synchpoint, as the APPLY program would have.

On the mid-tier server, each row has a timestamp property that indicates the last time the row was modified. Note the deleted rows are preserved with a "deleted" marker and a timestamp. When the client requests an update from the mid-tier server, the mid-tier server looks at the last time that the client was updated. Any rows which have changed or have been deleted since the last update are serialized to JSON and sent to the client.

B. UNTETHER *Change-Set Propagation*

Providing a persistent message queue on the device is not sufficient to implement the message-based approach for change-set propagation. Developers require, in addition, that these messages be removed from the queue and replayed on the server as an atomic operation. (We assume that a single message can be replayed on the server as a single transaction; i.e., UNTETHER does not deal with scenarios that require a two-phase commit [7] on the server.) UNTETHER is developing this transactional behavior using the approach of HTTPR [18] (for reliable delivery of HTTP packets) and WS-ReliableMessaging [19] (for reliable delivery of SOAP messages). These protocols ensure that all messages are delivered to their destination in their original form exactly one time.

As we develop applications on the UNTETHER infrastructure, we have found that the most obvious disadvantage of the message-based approach is its effect on the programming

model. A given piece of CWA business logic is coded exactly once: e.g., saving a PURCHASE ORDER either drives a service API *or* directly changes database state, but does not do both. With an ICWA using the message-based approach, we must code the same logic in two different forms. We save messages that will drive subsequent invocations of the service API to propagate the saved PURCHASE ORDER off the device. But, since the device must function while disconnected, this implies that the saved PURCHASE ORDER must *also* be saved in the device database in "raw data" form: e.g., to display the set of PURCHASE ORDERs that are pending approval at a given moment. While we have considered code-generation approaches to automate this dual code requirement, at this point, the application-specific nature of this code makes this difficult. Nevertheless, UNTETHER uses the message-based approach for the reasons discussed in Section II-B.

C. Enhancing Provisioning With Error Detection

UNTETHER also incorporates a mechanism to address a key issue for ICWA provisioning: namely, the difficulty in predicting the required data-set for a given application and given user (see Section II-A). If we define "correct provisioning" as ensuring that the result-set provided to a disconnected ICWA's query is identical to the result-set of a *connected* application's query, we can detect an incorrectly provisioned application with the following algorithm.

1) To implement a given ICWA query q, construct a query q_s (that queries the server's database) and a query q_c (that queries the device database).
2) At runtime, the ICWA executes q_c and uses the result-set of that query. However, the ICWA also stores the result-set returned by q_c (minimally, the set of primary-key values) as well as q_s (and its associated runtime parameters) for subsequent execution in a manner similar to the message-based approach for change-set propagation (Section II-B).
3) When the device connects to the server, the set of stored q_s queries are executed, and the result-sets compared to that returned to the original q_c execution. Differences between the result-sets implies an error in the original provisioning.

Care must be taken in step 3 to avoid incorrectly detecting an error in any of the following cases:

- If a datum was created on the device after the provisioning event, it is not an error if the result-set for q_c contains that datum. Similarly, if a datum was deleted on the device after the provisioning event, it is not an error if the result-set for q_c does not contain that datum.
- If a datum was created in the master database by another client after the provisioning event, it is not an error if the result-set for q_s contains that datum. Similarly, if a datum was deleted in the master database by another client after the provisioning event, it is not an error if the result-set for q_s does not contain that datum.

As with change-set propagation, UNTETHER does not generate the dual-query for the developer because (unless the device and master databases use the same API and schema) application-specific logic is required to code the different versions of the query. However, UNTETHER supplies the middleware to store the set of q_s invocations on the device together with the result-sets of the corresponding q_c execution.

IV. Conclusion

Web-browser support for recent HTML standards allow developers to assume low-level support for intermittently-connected web-application, without the need for additional plugins. Middleware support for transforming a connected version of the application to an ICWA is complicated by the need to incorporate application-specific logic throughout an ICWA's deployment. Our UNTETHER prototype instead focuses on providing middleware components for two problems: how to efficiently provision data onto the device, and how to propagate work performed on the device to the server. UNTETHER also incorporates a mechanism to detect logic and workflow errors in an application's provisioning algorithm. We are currently investigating whether analysis of a CWA's data-usage patterns can be incorporated into provisioning of the ICWA version of the application.

References

[1] "Offline web applications," http://www.w3.org/TR/offline-webapps/, 2008.

[2] "Indexed database api," http://www.w3.org/TR/IndexedDB/, 2012.

[3] J. Gehtland, D. Almaer, and B. Galbraith, *Pragmatic Ajax: A Web 2.0 Primer*. Pragmatic Bookshelf, 2006.

[4] T. O'Reilly, "What is web 2.0," http://www.oreilly.com/go/web2, September 2005.

[5] Wikipedia, "Representational state transfer," http://en.wikipedia.org/w/index.php?title=Representational_State_Transfer&oldid=109299419, 2007.

[6] M. Franklin, M. Carey, and M. Livny, "Transactional client-server cache consistency: alternatives and performance," *ACM Transactions on Database Systems (TODS)*, vol. 22, no. 3, pp. 315–363, 1997.

[7] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann, 1993.

[8] "Worklight jsonstore sync," http://pic.dhe.ibm.com/infocenter/wrklight/v5r0m5/topic/com.ibm.worklight.help.doc/devref/c_overviewofdatasynchronization.html, IBM, 2013.

[9] "Java Database Connectivity (JDBC)," http://docs.oracle.com/javase/tutorial/jdbc/, Oracle, 2012.

[10] "Java Naming and Directory Interface," http://docs.oracle.com/javase/jndi/tutorial/, Oracle, 2012.

[11] A. Leff and J. T. Rayfield, "Programming model alternatives for disconnected business applications," *IEEE Internet Computing*, vol. 10, no. 3, pp. 50–57, May/June 2006.

[12] ——, "Issues and approaches for web 2.0 client access to enterprise data," *Advances in Computers*, vol. 76, pp. 225–255, 2009.

[13] G. Harrison and S. Feuerstein, *MySQL Stored Procedure Programming*. Sebastopol, CA, USA: O'Reilly, 2006.

[14] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.

[15] K. Chodorow and M. Dirolf, *MongoDB: The Definitive Guide*. O'Reilly, 2010, iSBN:978-1-4493-8156-1.

[16] T. Hughes-Croucher and M. Wilson, *Node: Up and Running*. O'Reilly, 2012, iSBN:978-1-4493-9858-3.

[17] "Sql replication," http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=%2Fcom.ibm.swg.im.iis.db.repl.sqlrepl.doc%2Ftopics%2Fiiyrscncsqlreplovu.html.

[18] "Httpr specification," http://www.ibm.com/developerworks/webservices/library/ws-httprspec/, 2002.

[19] "Web services reliable messaging protocol," http://docs.oasis-open.org/ws-rx/wsrm/200702, 2005.