A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices

Yaron Weinsberg
Department of Electrical Engineering
Technion — Israel Institute of Technology
Technion City, Haifa 32000, Israel

Israel Ben-Shaul
Department of Electrical Engineering
Technion — Israel Institute of Technology
Technion City, Haifa 32000, Israel

E-mail: issy@ee.technion.ac.il

Abstract

The emergence of networked lightweight portable computing devices can potentially enable accessibility to a vast array of remote applications and data. In order to cope with shortage of local resources such as memory, CPU and bandwidth, such applications are typically designed as a thin-client thick-server applications. However, another highly desirable yet conflicting requirement is to support disconnected operation, due to the low quality and high cost of on-line connectivity. We present a novel programming model and a runtime infrastructure that addresses these requirements by automatically reconfiguring the application to operate in disconnected mode of operation, when voluntary disconnection is requested, and automatically resorting to normal distributed operation, upon reconnection. The programming model enables developers to design disconnected aware applications by providing a set of component reference annotations with special disconnection and reconnection semantics. Using these annotations, designers can identify critical components, priorities, dependencies, local component alternatives with reduced functionality, and state merging policies. The runtime infrastructure carries out dis- and re-connection semantics using component mobility and dynamic application layout. The disconnected operation framework, FarGo-DA, is an extension of FarGo, a mobile component framework for distributed applications.

The growing availability of networked lightweight portable computers (henceforth NLPCs), combined with the ubiquity of the Internet, has a great potential to enable accessibility to a vast array of remote applications and data. Since NLPCs are resource-constrained (small memory, CPU and low-bandwidth connections), distributed NLPC applications need to be designed with thin clients and thick servers that perform most of the computation and data access. Such design not only enables the execution of individual applications, but it also enables NLPCs to have access to more (client-side) applications with smaller footprint for each. Another requirement for NLPCs is the ability to operate in disconnected mode, due to the low quality and high cost of on-line connectivity. This is clearly a conflicting requirement, since disconnected operation requires that the NLPC will have local functionality and state that is sufficient for off-line operation. In order to reconcile these conflicting requirements, there is a clear need for a mechanism that can dynamically reconfigure NLPC applications when going from on-line to off-line operation and vice-versa. With proper policies that drive such mechanism, applications can leverage remote resources in the best way when connected, yet still run, perhaps with reduced functionality, in disconnected mode.

The dynamic reconfiguration mechanism should support several basic capabilities. First, it should support component migration and replication for downloading (uploading) components upon disconnection (reconnection). Second, it should support component replacement, i.e., the use of a component with reduced functionality but same interface, as a substitute to a (remote) component with complete functionality but large footprint, or as a replacement to an immobile component. Component replacement implies also the need for component reference replacement, i.e., the need to replace the endpoint of the component reference to refer to the new component. Third, it should provide means for an application to run even when not all of its components are loaded, and recover gracefully when an invocation is made on a non-existing component (e.g., when that component could not be downloaded due to space limita-
The general problem of NLPC support is very broad and complex. In order to narrow the scope of our problem domain, we make the following simplifying assumptions.

First, we distinguish between voluntary and involuntary disconnection. The former refers to a user-initiated event that enables the system to prepare for disconnection, and the latter to an unplanned disconnection (e.g., due to network failure). The main difference between the two cases is that in involuntary disconnection the system needs to be able to detect disconnection and reconnection, and it needs to be pessimistically prepared for disconnection at any moment, hence requiring to proactively reserve and obtain redundant (scarce) resources at the NLPC. Measures of fault-tolerance for NLPCs are an important topic in itself, but beyond the scope of this work. Here, we focus on voluntary disconnection, and on how to actually minimize the use of (scarce) NLPC resources.

Second, we reduce the set of supported applications to “uni-directional client-server”. That is, we assume that NLPCs act as active clients only, and are not servers of any interaction\(^1\). Furthermore, we assume that the disconnection request is always initiated by the NLPC (client), not by any server. The main reason for this restriction is cost-benefit. As shall be seen in Section 4, the problem becomes much harder and the solution more expensive when removing this restriction. At the same time, while this restriction might exclude some types of applications (e.g., peer-to-peer NLPC applications), the majority of future NLPC applications are likely to be uni-directional client-server, mainly due to the need to rely on stable and constantly available listening servers.

Third, as mentioned above, we adopt the distributed object programming model. As such, we assume that all interactions between distributed objects occur along (uni-directional) remote references between objects. This abstraction is essential to enable analysis and dynamic configuration of component interactions at the object (and reference) level, as explained below.

2.2 High-Level Approach

As an extension to FarGo, FarGo-DA uses the same two major abstraction vehicles for designing distributed and dynamically relocatable applications: composites and composite references. Composites are the basic building blocks of the application, somewhat analogous to CORBA [12] objects or RMI [18] remote objects, except that they also define the minimal unit of relocation. That is, a composite instance relocates in its entirety (unless otherwise specified, we will refer to composite instances simply as composites, for brevity). All objects within the same composite always share an address space and all intra-composite references are thus local.

Composites are interconnected via inter-composite references, henceforth termed composite references. Unlike remote references in conventional distributed frameworks (e.g., RMI), the same composite reference may be at times local and at times remote, depending on the (dynamic) relocation of its source or target composites during the lifetime of the applica-

\(^1\)To be more precise, we do not exclude cases in which NLPCs are servers, but FarGo-DA does not provide DA support for them.
tion. Unlike virtual references in other mobile frameworks which mostly provide (re)location transparency (e.g., Voyager [13]), complet references can be associated with rich semantics that describe various co- and re-location relationships between completes. Furthermore, these relationships are reified by the reference, and thus can be interrogated and evolve over time, e.g., to adhere to changes in the environment that demand changes in the relationships. (We defer a technical discussion of reference implementation to Section 4). For example, a Pull type of complet reference implies that when the source complet migrates, the target complet (and all of its Pulled completes, recursively) migrate along so as to preserve co-location between these completes. For a detailed discussion of relocation semantics see [5].

From the programming model perspective, FarGo-DA essentially adds a new kind of complet-reference types, termed disconnection-aware (DA), and few extensions to the complet abstraction. DA semantics define the behavior of completes with respect to migration and inter complet communication, before, during, and after a disconnection period.

FarGo-DA follows the principles that guided the original FarGo approach. Specifically, it preserves “syntactic transparency”, i.e., it requires no syntax changes in the programming language (standard Java), and no deviation from the Java programming model when coding individual completes. Moreover, the DA related semantics are encoded inside special reference types, and are thus separate from the rest of the application logic. This separation of concerns is important, since it enables developers to keep the application logic modules clean from DA issues, and address disconnection separately.

2.3 DA Complets and References

As mentioned above, FarGo-DA provides two main programming facilities for DA support: complet interfaces, and DA complet references. A complet is considered to be disconnection-aware if it implements the DA-Listener interface. This interface serves two purposes: 1. It defines DA related methods, preDisconnect and postReconnect, which must be implemented and invoked by the system during dis- and re-connection, respectively; and 2. It serves as a tagging interface used by the system at runtime to identify which completes are to be notified at runtime in order to carry out DA actions along DA references.

The first need for the above interface is clear: it enables the application to prepare for disconnection. An example use of preDisconnect is to generate and download remote completes which have not been instantiated yet, thereby allowing to migrate applications which are inactive at the time of disconnection. The use of the interface as a tagging facility requires justification. An alternative mechanism that does not require tagging could have been realized, in which the system scans all completes on the NLPC and notifies them when disconnection and reconnection events occur. The rationale for using a tagging interface is twofold. From the programming model viewpoint, it forces developers to explicitly state (and thus be aware of) which completes are DA, and it enables the compiler to identify these references and generate the code for them. From the system viewpoint, an added value is that only DA completes are notified, avoiding unnecessary notification of (non disconnected-aware) completes at runtime.

As for complet references, FarGo-DA provides a host of references with semantics that address the requirements set forth in Section 1. The programming interface for setting these references is quite simple, as shown in Figure 1. The reference is reified using the getMetaRef system method, and then assigned the proper semantics (the actual parameters of setDA method are explained below). Notice that since the reference type is assigned dynamically, it can also be modified at runtime. We shall see in Section 3 the usefulness of dynamic reference types.

```java
...
// get a handle for the reference from
// this object to the target complet
MetaRef mr = Core.getMetaRef(target);

// set DA semantics on the reference
mr.setDA(new Clone(),
      new Merge(),
      Priority.HIGH);
```

**Figure 1. Setting DA semantics on Complet Reference**

FarGo-DA provides a set of DA reference types, some of which apply only to disconnection, some only to reconnection, and some to both. This set of reference types is by no means intended to be exhaustive, however, and we expect to add more types as practical need for them arises. We focus on the semantics here, and defer technical aspects of the implementation of these references to Section 4.

- **Clone** – This reference denotes that the target complet is to be duplicated and migrated to the local NLPC prior to disconnection from the server, so as to enable continuation of the application execution. The source complet is not affected by the cloning, in that it continues to execute and refer to the cloned complet exactly as it used to refer to the original (remote) complet, transparently. The target complet’s cloned copy has the same state as the original complet prior to disconnection. Clone is transitive, i.e., if the target complet has other clone references, they are also cloned, recursively. Using this reference, the programmer can distinguish between completes that are essential to the application and thus should be cloned, and non-essential completes which should not be cloned.
- **Replace** – This type is used to replace a reference to a remote complet with a temporary reference to a local complet. Upon disconnection, a local complet is looked up and gets connected to the source complet. If no live complet instance exists, a new one is created.

---

\(^2\)We use the common terms source complet and target complet to denote the referencing and referenced completes, respectively.
The local complet should have the same interface as the remote complet, but may have a different implementation. A Replace reference is most useful when a local complet with reduced functionality (and hence smaller footprint) can temporarily replace the original remote (and possibly too large) complet. Another use is when the origin complet cannot be migrated, because it relies on non-migratable resources such as a large database.

- **StoreAndForward** – All invocations that are made along this reference after disconnection are stored at the local NLPC, pending for reconnection. These invocations must be one-way (i.e., return immediately with no return parameters) in order to enable the source complet to continue operation. Upon reconnection, all stored invocations are carried out against the remote core. An example use of this reference type is for outgoing email messages.

- **Depart** – This reference operates in an opposite manner to StoreAndForward. All invocations that are made along this reference prior to disconnection are handed over to the remote core for execution. Thus, from the source complet viewpoint, this is an asynchronous invocation that does not require the link to be alive constantly. Upon reconnection, the results, if any, are transferred to the source complet. The source complet can either poll or get notified when the results become available. When in disconnected mode, this reference acts as a StoreAndForward reference. The motivation for this reference is to spawn (or batch) heavy remote computations without the need to be continuously connected, and receive the results automatically as soon as reconnection occurs.

- **Reconnection Types**

  FarGo-DA provides four types of DA references that dictate the behavior of the referencing and the referenced comlets upon reconnection, with respect to merging the state of cloned and replaced comlets. The motivation for these types is to address the cases where, after disconnection, both the origin and cloned (or replaced) comlets change their states independently. Reconnection types are specified as an additional parameter to the setDA method (see Figure 1).

  1. **Purge** – This type implies that any state that was changed on the cloned or replaced complet is overwritten by the original complet. It is similar to changing the state of an object that was passed by value. The object is used to invoke its methods and to read its state, but not to modify its state.

  2. **Overwrite** – The opposite of Purge. The state of the cloned complet overwrites the state of the original complet.

  3. **Last** – The state of the complet with the latest timestamp wins. This type may be used only with comlets whose modification times are recorded (e.g., a persistent complet whose serialized state is kept in the file system), and in cases where the clocks of the NLPC and the remote server are synchronized.

  4. **Merge** – Neither the cloned/replaced complet nor the original complet clearly "win", and there is a need for conflict detection and resolution techniques, which are elaborated below.

2.4 Conflict Resolution Methods for Merging Complets

It is generally impossible to devise global applicable conflict resolution strategies that could resolve conflicting updates between any replicas without assuming application specific knowledge. FarGo-DA offers two mechanisms for applications in order to resolve merging conflicts: callback methods and reconciliation operators. The former is straightforward. Any disconnected-aware complet can implement the interface **MergerInterface**. This interface consists of a **Merge** callback method that the application developer must implement. At runtime, the system activates this method upon merging the cloned complet with the origin complet. To further assist in the merging process, FarGo-DA provides several facilities. First, it can log all invocations along a complet reference that are made on both the cloned complet and on the original complet (by other users who are still connected to the original one). Second, it can keep a separate archive copy of the cloned object with the state it had right after disconnection. This is useful since, as mentioned above, the original complet might have also been changed during the disconnection period. Finally, when reconnection occurs, the **Merge** method is invoked on the cloned complet, accepting as parameters the archive, the original complet, and both log files. Figure 2 illustrates the use of this method.

```java
void Merge(Compleat archive, Compleat original,
           Log localLog, Log remoteLog)
{
  // Application-specific merging code
}
```

**Figure 2. Application Specific Merging Methods**

In addition to the above manual (from the programmer's perspective) merging procedure, FarGo-DA provides a semi-automatic merging process through the use of reconciliation operators. The general idea is to encapsulate Java primitive types within classes that contain, in addition to actual primitive values, a state variable that represents a built-in merging method. For example, Figure 3 shows the instantiation of an object from the (system supplied) class **disInt**, with add as its merging operator. Using these primitive type wrappers, FarGo-DA can support automatic merging of any user-defined types as follows. Each class that is subject to such merging, should implement the
2.5 Complet Prioritization

An important aspect of DA support for resource constrained devices is complet prioritization. Since storage (and bandwidth) constraints might prohibit the downloading (or the local generation) of all completes that were scheduled for download in disconnected-mode, there is a need to enable developers to assign complet priorities that determine the downloading order. FarGo-DA provides a simple interface to setting priorities, by an additional parameter to the setDA method (see Figure 1). At runtime, the system carries out the disconnection semantics based on the priority order. A potential extension of this mechanism is to prioritize whole applications. However, it is important to realize that while complet priorities should be defined by the developer who understands the semantics of the application, NLPC-level priorities should be defined by the end-user who uses the applications, and require therefore a different, application-level, external interface (not provided currently by FarGo-DA).

2.6 Graceful Recovery from Missing Complets

Recall that an important consideration in supporting disconnected-aware applications on NLPCs is graceful recovery when not all components exist in the NLPC. Consider, for example, the following scenario: \( \alpha \xrightarrow{\text{Clone}} \beta \xrightarrow{\text{Delete}} \gamma, \) with \( \alpha \) residing in the NLPC and \( \beta \) and \( \gamma \) in the remote server. Upon disconnection, \( \beta \) would be cloned to the NLPC, but \( \gamma \), which is not essential for the application, would stay at the remote site. If \( \alpha \) invokes a method that ordinarily would propagate to \( \gamma \), the system needs to be able to recover from this state. One simple solution is to rely on Java's exception mechanism. Since complet references are implemented on top of Java RMI, with no special handling, the above scenario would lead to the application throwing the \text{java.rmi.RemoteException}. In order to address specifically disconnection related faults, the FarGo-DA runtime converts this exception into proper \text{DisconnectedException}. This exception should be handled by the "root" of the complet (termed anchor), which must be on the call stack, and hence eventually catch the exception.

The main drawback of this standard approach is that it forces the application to treat all complet misses as Java exceptions. While this should be the alternative of choice when a missing complet is indeed critical for the execution of the application, it is desirable to provide means that will enable an application to select another method to invoke when one is not available (due to a complet miss) as a normal control-flow, as opposed to exception control flow. FarGo-DA supports such capability via its invocation interception mechanism (whose details are given in Section 4), already in place for general support of complet references. Before handing the invocation to the target complet (but only when in disconnected-mode), the interception layer checks if the target complet exists. If it does not exist, then the \text{missingComplet} system method is invoked. The system-supplied default method pops up an interactive window that identifies the type of the missing complet and the method signature used in the invocation, along with two buttons: \text{abort} or \text{continue}. If the user selects the latter, then the invocation returns to the caller method, who can now continue execution (e.g., by invoking an alternative method on another resident complet). Otherwise, if the user elects to abort the application, the \text{DisconnectedException} exception is thrown to the caller, as explained above. Finally, the \text{missingComplet} method can be overridden by the application programmer on a per-complet basis, to enable user-defined recovery.

3. Example Application

Let us illustrate the programming model by presenting an application that was actually implemented in FarGo-DA. The application is intended to be used by remote sales forces. Each sales agent is equipped with an NLPC. The NLPC may at times be connected to the central facility, and at other times disconnected from it. Connectivity may be subject to cellular coverage in the visited area, the cost of connection, and so forth. The central facility maintains information about customer accounts, product inventory, and it stores tools and data for calculating the product that best matches the customer's needs based on input gathered from the customer and from the central facility.

A typical operation of the client application by a sales agent involves: 1. overview of available product types and analysis of the most suitable product; 2. Browsing the customer accounts; and 3. Making an order against the inventory database. A major design goal is for the application to be disconnected aware. Thus, when connected, it should operate effectively by leveraging access to remote resources, yet when disconnected, it should also be able to work in the best possible way despite the lack of connectivity and short-
age of computing resources. Finally, when reconnecting, it should smoothly switch from disconnected to connected mode, by propagating work that was done locally to the central server, and by resorting to using remote resources effectively.

In the sequel we show how, by using FurGo-DA, developers can design and implement such application with a high-level of disconnected-awareness, yet with relatively small efforts. For brevity, we only focus on the main and on DA-related functionality. For more details on the system implementation see [19].

3.1 Solution Design

The application consists of the following complets:

- **GUI** – This complet serves as the user interface of the application. It does not contain any application logic. It merely translates actions performed by the user to invocations on the Logic complet. This complet always resides at the NLPC.

- **Logic** – A central complet in the application, which handles all service requests, possibly using other complets.

- **Inventory** – Maintains product inventory and accepts delivery orders from sales agents. Each product is represented by a separate inventory complet.

- **Accounts** – Maintains customer accounts. Each customer is represented by a separate complet.

- **Configurator** – This complet contains logic that generates the best product for a given customer, based on the customer input and on search in the customer database.

- **BasicConfigurator** – This is a low-end utility that creates a sub-optimal product selection solely based on input from the customer. It typically resides with the GUI complet at the NLPC.

Under connected mode of operation, the sales agent invokes the application through the local GUI complet, which is connected to the remote Logic complet that resides at the server. Using the GUI, the agent can access accounts, inventory, and configurator tools.

The interesting design choices are made when disconnection and reconnection modes are considered. The resulting DA-aware design is shown in Figure 4, and the complet layout when in disconnected mode is shown in Figure 5. The design choices are mostly reflected by the reference types, and we elaborate below on some of the choices made.

The Logic complet must be cloned to the NLPC in order to perform any meaningful activity; it does not generate meaningful output, and it is an essential element in the application. Hence, the resulting DA type on the GUI to Logic reference is (Clone, Purge, High). The Configurator tool should be replaced by the local low-end BasicConfigurator utility in disconnected mode, and no input is generated, hence the Replace, Purge, High reference type. Account management may be optionally accessible, although

---

Figure 4. Application Layout, Connected Mode

Figure 5. Application Layout, Disconnected Mode

379
when visiting an existing customer, its record should be cloned. Upon reconnection, the new customer record should be merged with the remote customer account. The resulting reference type is then (Clone, Merge, Low). In order to address the special need to download specific account complet as a function of the sales agent’s location, we use the dynamic reference configuration capability of FarGo-DA. That is, upon disconnection, the preDisconnect method in the Accounts complet looks up for a customer account that matches the currently visited Customer, and if found, sets its priority to High, thereby increasing the likelihood of cloning that complet. Finally, the Inventory complet need not be available when in disconnected mode, but orders should be propagated to the inventory manager as soon as reconnection occurs. Thus, it is connected by a storeAndForward reference type. When disconnected, orders are queued locally. As soon as reconnection occurs, orders are executed against the inventory.

In order to give an idea of how DA primitives are coded in the application, we provide some sample code fragments in the Appendix. For more details on this fully implemented application, see [19].

4. The Runtime Infrastructure

4.1 Architecture

FarGo’s runtime infrastructure is composed of a collection of distributed stationary components called Cores. Each Core runs within an instance of a single Java Virtual Machine (JVM), which in turn runs inside a single operating system process.

The architecture of a single Core is shown in Figure 6. The DA Reference Handler implements the complete reference interception layer. The Movement unit migrates/accepts complets. The Invocation unit implements the parameter passing scheme over complete references. These three entities realize most of the runtime support in FarGo-DA. The Complet Repository stores complet references and is used by the Naming service to map logical names to complets, and the Peer Interface layer performs low-level Core-to-Core communication.

In order to provide a lightweight Core that can run on small devices, some of the FarGo components were omitted in the Cores that run on the NLPCs, including the monitoring facility (outside the scope of this paper, see [3]) and all system complets (including the shell). The main additions to the original architecture of FarGo are the Disconnected Manager (DM, explained below), the DA-referencing layer and the DA API.

The Core API provides various services to applications, including initial activation of the Core, manipulation of complet references, movement, naming, monitoring, and disconnection and reconnection methods. Programmers use both the Core API and the standard Java API to implement their application’s complets.

A unique aspect in FarGo is the design of complet references. Typically, a remote reference is implemented by a regular local reference from the source to a proxy object, which in turn points to the remote object (through some system-generated code). In FarGo, a complet reference is actually divided into two separate entities: a stub and a tracker. The stub object is pointed by the source object using a regular local reference and its interface is identical to the interface of the target complet. The tracker is responsible for tracking the target complet and for actually performing invocations. There are two advantages to this separation. First, the stub always refers only to a local object. In particular, whether the target complet is local or arbitrarily remote, this has no effect on the stub. Thus, the stub’s interface can be nearly identical to that of the target’s anchor, because no remote code need to exist in it. This design facilitates transparency, since the programmer can access the stub exactly as if it were the target anchor. Second, the addition of a level of indirection allows to keep only one tracker per target complet in a single Core, although the number of complet references that point to this target complet can be large.

Figure 7 illustrates the internal structure of complet references. The left side shows two complets, α and γ, both pointing with complet references to a third remote complet, β. The right side of the figure shows a closer look at these references. The tiny diagrams inside α and β represent intra-complet objects that are part of their closure. β’s stub has an outgoing regular (local) reference to the (local) tracker, for passing all invocations, and a reference to a meta-reference object who is used for reifying the references.

The tracker(s) are responsible for achieving location transparency (as in [2, 17]). Upon the arrival of a complet to a new site, a new tracker is generated there and is set to directly point to that complet. Then, the tracker at the old site is set to point to the tracker at the new site. After several hops, a chain of trackers is being formed, and
each tracker forwards invocations to the next one until the target’s anchor is reached and invoked. Chains are automatically shortened by the runtime.

4.2 Disconnection Management

FarGo-DA's infrastructure extends FarGo by adding a runtime disconnection (and reconnection) management, and functionality that realizes the DA reference semantics. In addition, the FarGo compiler has been extended to support the generation of proper system calls that correspond to DA references.

Disconnection Management (DM) is responsible for carrying out disconnection and reconnection in the system. It is implemented as a distributed subsystem. Each core hosts at most one DM, which is dynamically loaded only when a disconnection event arrives (as a result of an explicit user request), and is unloaded after reconnection has completed (loading/unloading of complements is meaningful when the NLPC has both RAM and disk, in which case the DM needs to be resident in memory only during disconnected-operation).

Upon the arrival of a disconnection event, the Core first fires an event to all local disconnected-aware complements (i.e., complements that implement the DA-Listener interface), which in turn invoke their own preparation methods. The next step after all local preparations are done, is to traverse the reference graph. Using Java introspection, each DA complemt (and each of its contained objects, recursively) is inspected for outgoing complement references. The references are sorted by their (user-defined) priority and stored in a priority queue. Then, each reference is handled according to its specific DA semantics. Notice that the DM must maintain sufficient state on each reference in order to be able to restore the system to its state before disconnection.

Clone references are handled as follows. The DM requests from the remote Core in which the target complet resides, to duplicate the complet. The complet then migrates, and gets immediately introspected, storing the newly discovered references in the priority queue. Thus, with respect to Clone references, a depth-first traversal is performed based on the priorities of the references.

It can be seen now why supporting bi-directional client-server applications is hard. FarGo-DA bases its DA support on the reflective capability to traverse and discover outgoing references. There are no means to identify, for a given complet, what are its incoming references. Such capability would have been required in order to support bi-directional disconnection. A brute-force approach to discovering incoming references into complements in Core A could be implemented by requesting all remote Cores in the system to traverse their local complements, search for those who reference complements in Core A, and then apply on them DA actions. However, this is an expensive solution that might lead to long pre-disconnection periods, and thus not appealing. Hence the limitation to uni-directional applications.

As for Replace references, DM looks up for a local replacement complet with the proper interface (using the Core, which maintains a (weak) reference to all hosted complements). If found, the reference is changed to point to the new complet. If none is found but the class exists locally, a new instance is created. Otherwise, an exception is raised. For both Clone and Replace, the id of the original complet is kept in the DM database, for reconnection purposes.

The StoreAndForward reference requires minimal preparation work during disconnection, since there is no need for cloning or for changing existing references. Most of the semantics are employed when such reference is actually invoked.

The implementation of Depart is more complex. The

---

4Both state and classes are migrated, unless the classes already reside at the NLPC; for more details on FarGo-DA’s class loading mechanism, see [15].
general idea is to pass the invocations to a newly created thread that resides at the core of the target completion, and is ready to invoke the methods locally.

4.3 Reference Support During Disconnection

Once cloned, invocations along a Clone reference require no special support; they are treated as normal completion references, except they are local, hence optimized to use local references (see [6] for optimizations to completion references). Similarly, Replace completes require no special treatment. StoreAndForward references are handled by accumulating invocations (including method names and actual parameters) in a local store. Invocations along a Depart reference are disallowed, since by definition the two ends of the reference are disconnected. However, the remote proxy thread invokes the requested methods locally and accumulates the results. Finally, as mentioned earlier, FarGo-DA provides optional logging facilities for all completion references, regardless of their type.

4.4 Reconnection Management

Upon reconnection, the disconnection manager is responsible to restore the connections that existed before disconnection. Clone and Replace reference types are handled depending on the defined reconnection semantics. For lack of space, we omit a discussion of this procedure. StoreAndForward references are handled by the system using reflection. The disconnection manager looks up the target completion (using search by id on the remote core), looks up the proper method in that completion, and invokes it with the input parameters. Finally, Depart reconnection involves getting the return values from the remote thread and returning them to the source completion. In general, this is done by creating another proxy completion that accepts the return values from the remote completion, and makes them available locally using polling or via event notification. As mentioned above, Depart implementation is quite complicated, and it is actually based on an earlier implementation of asynchronous invocation in FarGo, see [1].

5. Related Work

Much of the early work on support for disconnected operation has been in the area of distributed file systems. Two examples are Coda and Ficus.

Coda [10, 16] is a distributed file system that extends AFS [7] with additional support for high availability. Disconnection is treated as a special case of network partitioning. A disconnected client can continue working by using any data it has in its cache. Information from the user’s profile is used to help keep the best selection of files in an on-board cache. While disconnected, Coda’s client services file system requests only from its cache, treating cache misses as failures (i.e., file not found errors from the file system). When the network reconnects, the cache is automatically reconciled with the replicated master repository. Coda provides automatic conflict resolution mechanisms for directories, and uses Unix file naming semantics to invoke application-specific conflict resolution programs at the file system level.

Ficus [14] is another distributed file system that supports a form of disconnected access. Ficus employs an optimistic replication which allows to update local copies and detects (rare) conflicts at reconnection time. Furthermore, Ficus uses its knowledge of the semantics of updates to resolve most of the conflicts automatically.

The main focus of these projects is on high degree of (data) availability in the face of unplanned disconnection. None of these projects addresses the software engineering aspects of constructing applications that need to operate under such conditions.

Thor [4, 11] is a distributed object-oriented database (OODB). Applications interact with Thor by invoking object methods and requesting transaction commits. Persistent objects reside at server machines and applications run at client machines. Copies of persistent objects are cached on clients to reduce delay for applications and offload work from servers. While a client is disconnected, it can continue to run if the needed objects are in its cache, but will be unable to make progress (at least in doing that particular task) if objects are missing. Each method call (made by the client application) that led to the cache miss fails with an exception indicating that a cache miss was the problem. (all method calls to Thor can terminate with an exception). The application can respond properly, e.g., by aborting the current transaction and suspending with a message to the user indicating that it cannot proceed until a reconnect occurs. Besides invocation on cached objects, Thor does not provide mechanisms for implementing disconnected-aware applications (e.g., non-blocking invocations), let alone programming facilities for defining disconnected-aware semantics for applications.

Rover [8, 9] is the most related work to FarGo-DA. It provides a software toolkit for construction of mobile-aware applications. The Rover toolkit provides two major programming abstractions: relocatable dynamic objects (RDOs), and queued remote procedure call (VRPC). A relocatable dynamic object can be dynamically loaded into a client computer from a server computer. Queued remote procedure call is a communication system that permits applications to continue to make non-blocking remote procedure calls even when a host is disconnected, with requests and responses that are queued and exchanged later upon network reconnection. Rover maintains a cache of RDOs, and provides facilities to detect conflicts between cached and “origin” copies, upon reconnection. VRPCs are similar to StoreAndForward references in FarGo-DA. But unlike Rover, FarGo-DA provides various alternative reference types that can implement additional semantics such as Clone, Replace and Depart, and those semantics may even change dynamically depending on the application’s state and needs. Another major difference between Rover and FarGo-DA stems from the latter’s focus on resource-constrained devices. In FarGo-DA there is no built-in cache. Thus, when not disconnected, there is no (space and time) overhead for maintaining the cache. While a caching mechanism is indeed essential for supporting involuntary disconnection, it might be prohibitively expensive when deployed on a resource-constrained device. Instead, FarGo-
DA's Clone reference may be viewed as a "cache-on-demand" utility, which becomes active only during disconnection, and automatically disappears upon reconnection, freeing up precious resources on the NLPC when caching is not needed.

6. Conclusion

FarGo-DA proposes a new dimension of flexibility for architects of large-scale distributed applications, namely support for designing the behavior of applications under frequent disconnection conditions. The programming model enables designers to augment their applications with disconnected aware semantics that are tightly coupled with the architecture, and are automatically carried out upon disconnection.

7. Acknowledgments

This work has been partially supported by a grant from Microsoft research. We thank present and past members of the FarGo project, including Miki Abu, Tamir Ronen, Ophir Holder and Hovav Gazit, who have contributed to the FarGo research in general. Special thanks to Benny Reich and Ophir Holder, who have advised us about the example application.

8. REFERENCES


APPENDIX

A. Solution Implementation

The following Figures below show fragments from the implementation of the example application. Figure 8 shows the creation of the outgoing references from Logic, with proper DA semantics. Figure 9 shows how the Account complete upgrades, during disconnection, the priority of the reference that points to the customer that is visited, at time of disconnection, by the requesting sales agent. This priority upgrade ensures that the desirable customer account would get downloaded to the agent’s NLPC.

```java
public class Logic_ extends Complet {
    private Accounts accounts;
    private Inventory inventory;
    private Configurator calc;

    public Logic_()
    {
        MetaRef mr = new null;
        accounts = new Accounts();
        mr = Core.getMetaRef(accounts);
        mr.setDA(new Clone(),
                new Merge());

        inventory = new Inventory();
        mr = Core.getMetaRef(inventory);
        mr.setDA(new StoreAndForward());

        calc = new Configurator();
        mr = Core.getMetaRef(calc);
        mr.setDA(new Replace(),
                 new Purge(),
                 Priority.MEDIUM); // default
    }
    // the rest of the complet ...
}
```

Figure 8. Setting Static Connection Semantics

```java
public class Accounts_ extends Complet {
    private HashTable accounts;

    public Accounts_(String[] args) {
        accounts = new HashTable();
        // ... read the accounts from a DB
        // and put them in the HashTable
    }

    public void pre_disconnect(String current) {
        // locate the current costumer reference
        // set the priority to high to increase
        // the chances to clone it...
        CompanyAccount ac = null;
        MetaRef mr = null;
        ac=(CompanyAccount)accounts.get(current);
        mr=Core.getMetaRef(ac);
        mr.setDA(new Clone(),
                 new Merge(),
                 Priority.HIGH);
    }
    // the next methods ...
}
```

Figure 9. Dynamically changing the connection Semantics