Footloose: A Case for Physical Eventual Consistency and Selective Conflict Resolution

Justin Mazzola Paluska   David Saff   Tom Yeh   Kathryn Chen
MIT Computer Science and Artificial Intelligence Laboratory
{jmp,saff,tomyeh,kchen25}@mit.edu

Abstract

Users are increasingly inundated with small devices with communication and storage capabilities. Unfortunately, the user is still responsible for reconciling all of the devices whenever a change is made. We present Footloose, a user-centered data store that can share data and reconcile conflicts across diverse devices. Footloose is an optimistic system based on physical eventual consistency—consistency based on the movement of devices—and selective conflict resolution—which allows conflicts to flow through devices that cannot resolve the conflict to devices the can. Using these techniques, Footloose can present consistent views of data on the devices closest to the user without user interaction.

1. Introduction

The way in which many people kept the data that is important to their lives changed with the introduction of the personal computer, which offered a centralized repository for data that might otherwise be scattered in file folders, address books, or just a person's memory. In the last few years, there has been explosion of personal peripherals, including PDAs, digital cameras, cell phones, MP3 players, laptop and palmtop computers, wireless pagers, and more. Each of these devices is synchronized regularly with the "Home PC". Hence, the Home PC is responsible for making sure that the data from a peripheral is both persisted and consistent with other devices in the system. It ensures that working with the peripheral is the same as working with any other device in the system.

We believe that current trends show that as the number and variety of mobile peripheral devices grow, and as expectations of mobility increase, it will no longer be adequate to expect for consistency to be maintained by one-to-one synchronizations directly with a Home PC. Rather, we envision that the device communication topology will look much like Figure 1. Wireless communication standards like Bluetooth will allow a cell phone, PDA, and laptop to talk directly to each other, and users will expect this bandwidth to be used even when they are away from their Home PC for extended periods of time. A computer at the user's work, and perhaps even in her car, may never come into direct network contact with the Home PC. We also imagine a market for devices like what we call a BlueBox (such as HP Labs Personal Server [8]), a Bluetooth-enabled large storage device small enough to be carried everywhere, and with limited processing capability.

All of these trends indicate that consistency will have to be maintained through the one-to-one interactions of many devices, of which the Home PC is just one. We present the design and initial implementation of Footloose, a user-centered data store that manages data across diverse devices. Footloose is aimed at applications centered around a single user. Data stored in Footloose can be modified by any device the user owns . Furthermore, Footloose ensures that the changes are reflected in all devices interested in the change see the change as long as long as they communicate with at least one other device that knows about the change. Correspondingly, devices uninterested in the change pass the data along in order to help the interested devices converge on a single data state.

Footloose introduces two ideas to maintain the user's data store:

1. *physical eventual consistency* on a human time-scale, and

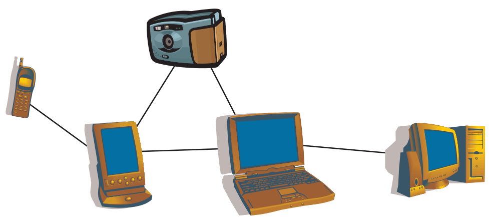

Figure 1. A typical set of devices and communication pathways

2. application-based *selective conflict resolution*.

Physical eventual consistency is a protocol that uses physical proximity of devices to enhance consistency. Application-based selective conflict resolution allows applications distributed among *some* of the devices to communicate, even if at any given point, two devices disagree on a piece of data. It is also the enabling technology behind physical eventual consistency. Together, the two methods allow devices to carry all kinds of data—consistent, inconsistent, and tentative—for all applications on all devices without restraint.

2. Environment and Design Strategy

Two scenarios give an overview of the kind of interactions we expect in a device-oriented world and want to support in Footloose. These will also help explain the particular choices we made.

In the first scenario, Alyssa calls Ben before work on his cell phone with her new number. His phone memorizes the phone number using caller ID and stores it in its phonebook. As Ben gets ready for work, he puts his cell phone near his briefcase with his PDA. The cell phone notices that the PDA is within wireless range and passes on Alyssa's new number to the PDA before Ben takes his briefcase to work while leaving his personal cell phone at home. Once at work, Ben leaves his PDA on his desk. Again, it finds that his computer is in wireless range and further passes on the new number to his desktop personal information manager (PIM). Here, all of the devices deal with contact information so it is consequently reasonable for them to synchronize themselves when they come in contact.

In the second scenario, Louis purchases an item online at work and enters his purchase in a financial planner. His work computer knows that his home computer keeps track of this information for Louis and would like to propagate this data to it. However, Louis home computer is off and cannot be reached. Fortunately, Louis's cell phone is within wireless range of the work computer. The work computer connects to the cell phone and determines that it is likely to see Louis's home computer, so the work computer transfers the purchase information to the cell phone. When Louis returns home that evening, as Louis checks his e-mail, the cell phone in his pocket comes within wireless range of the desktop and transfers the new purchase to the home computer. Here, the cell phone acts merely as a carrier of financial information it has no need or ability to understand.

The first scenario is somewhat possible using today's tools [19, 7, 20]. However, a user must manually initiate synchronization or manually copy data. Footloose makes this completely automatic. The second scenario is not possible with current devices—most small devices only store data they understand and have no way to act as a carrier for other devices. Footloose enables the second situation, again, in a completely automatic manner.

2.1. Design Assumptions

An obvious problem in the above two situations is keeping data consistent among the various devices. Consistency among distributed hosts is not a new problem. Historically, replicated file systems have attempted to maintain user data with varying degrees of consistency [6, 5, 11]. Some systems have also tackled other kinds of data stores, such as calendars, with varying degrees of success[17, 2]. A common thread among replicated file systems—and especially those geared toward mobility—is *optimism*. Such systems allow anybody to make updates at any time and reconcile conflicts later.

Since we want to enable the user to make updates at any time, Footloose is an optimistic system. However, devices in Footloose live a different environment than the clients of any of the above systems. Specifically, we assume that:

1. Connections between devices are fleeting, but form a connected graph over time.
2. Applications run on many devices that may not directly talk over time.
3. Not every device has the resources to understand all of the data types in the system or resolve conflicts.
4. Devices have a finite amount of storage.

We envision a typical inter-device connection to be like that of a Bluetooth "personal area network." Furthermore, we target the class of applications where the user is the main "writer" or updater, such as PIMs. In return, Footloose will provide a guarantee of "no lost updates" to an interested distributed application and eventual convergence on a single data state.

The first two assumptions and the target application class led us to choose a variant of eventual consistency we term physical eventual consistency to manage our data. The latter two assumptions helped us reason about our data management and routing protocols and also our conflict resolution algorithms.

2.2. Physical Eventual Consistency

As both scenarios show, Footloose works in a primarily disconnected network environment. Therefore, to make any useful progress, it is obliged to make optimistic updates. The associated consistency model is *eventual consistency*—data will converge on a single state given enough update messages passed among participating devices [17]. A major detraction of optimistic systems is that there is often no bound on the amount of inconsistency in the system at any given time.

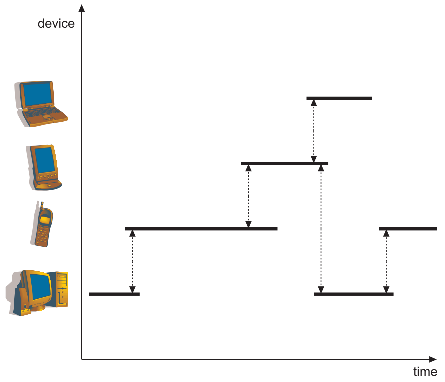

Figure 2. At any given time, the horizontal bars represent devices near the user. Synchronizations are the vertical dotted lines. As long as there is a data path between devices over time, Footloose will maintain data consistency.

However, we argue that for user-centered applications, consistency is closely related to how a user interacts with devices in the system and what inter-device interactions the user allows. As devices become "digital jewelry" worn by users, interactions are limited by how physically close two devices come. No matter how large the number of devices gets, the human user will still be the center of the system and data consistency is maintained for her sake. The user is also the primary source of updates. Together, these observations mean that as consistency is maintained in an ad-hoc way between devices, the device *closest to the user* will contain the most up-to-date data and hence will have the lowest amount of inconsistent data.

Therefore, we claim that as long as

1. at any given time, one device is closest to the user and
2. the outgoing closest device and an incoming closest device can communicate,

a user's data and updates can be maintained *perfectly* using eventual consistency. These claims are analogous to a relay race where the baton is a particular update and the racers are devices. In the relay, the baton is usually close to one racer except for brief transfer points, but in the end makes it all the way across the track to the final runner. In Footloose, we allow the device carrying the data to change and guarantee that the updates to data will propagate through all of the devices.

For example, Figure 2 shows devices close to the user over some time period and relevant synchronizations that propagate state. The figure is a connected graph over time—it is possible to reach any of the devices from the initial starting point by traveling forward (right) in time or up and down between devices. As long as the user keeps any one of her devices physically with her at any given time, then given our claim that the outgoing closest device and incoming closest device can communicate, a connected graph is formed. On such a graph, an update on one device will travel through the user's devices and eventually arrive at her interested devices.

Unfortunately, sometimes relay racers drop their batons and, correspondingly, sometimes incoming and outgoing closest devices do not always have the chance to fully communicate changes. For example, if Ben left both his PDA and his cell phone at home, Alyssa's number does not get propagated to Ben's office computer. In this situation, none of the updates on the home devices could have had the chance to propagate to the home devices—the data baton was dropped. What is worse, is that Ben could change Alyssa's number on his office computer, potentially leaving two new versions of Alyssa's number among his devices. In terms of Figure 2, we have a disconnected graph of devices and there is no way to get to certain devices by going forward in time. In this case, Footloose may not present a perfect view of the user's data, but still allows updates to be made. Going back to Ben's scenario, in the likely event that Ben brings either his cell phone or his PDA to the office or vice versa, it will bring from home all of the home updates and bring back home all of the office updates, allowing Footloose to propagate Alyssa's number and reconcile any other differences among all of the devices. Hence, after this step, Footloose can continue normally, with a reconnected graph.

We detail the algorithms for propagating device data in Section 3.

2.3. Selective Conflict Resolution

Optimistic systems are useless without methods to resolve conflicts. Indeed, the forgetful Ben of the previous paragraph may have changed Alyssa's number differently on the temporarily disjoint sets of devices. In the distributed file system world, there are only a few data types (files, file metadata, and directories) and every participant in the system understands these types, so it is reasonable to expect all participants to be able to file conflicts. Bayou [2] extends this and allowed arbitrary types to be resolved based on application preferences. However, Bayou requires resolution abilities at every node since conflicts must be resolved as soon at they hit the Bayou network. Unfortunately, we do not assume that Footloose's devices have the power to resolve such conflicts.

Instead, we use *selective conflict resolution*. Devices in Footloose may be computationally weak, so we divide devices into two broad classes, *smart* devices and *dumb* devices. This classification is on a per-application basis, so what may be a smart device to one application can be a dumb device to another and vice versa. Smart devices have an application conflict-resolver and are able to resolve conflicts between updates. Dumb devices cannot resolve conflicts. However, dumb devices can store and forward data in the hopes that a smart device will resolve the conflict. In the Alyssa and Ben scenario, all of the devices are smart since they all have contact information applications and could be called upon to resolve a contact information conflict. In Louis's situation however, his two computers are smart, but his cell phone is dumb since it can only act as a shuttle of financial information. The applications installed on a device determine the selection of conflicts it can resolve. All devices can let all other updates to opaquely flow through the system.

2.4. Footloose

Footloose ties together physical eventual consistency and selective conflict resolution with routing heuristics to provide an environment where applications can share data among many devices. Specifically, Footloose gives applications the Footloose Store (FLS), a typed key-value map in which to store shared data. This structure is general enough to allow applications to build their own data types though we also provide synchronization callbacks if applications want to use their own data structures. Footloose takes care of moving the data inside the FLS between devices and notifying applications when there are conflicts. Applications merely have to use the store and be able to handle conflicts if they want to use Footloose.

The two sections that follow describe the architecture of Footloose in more detail and evaluate the implementation of Footloose in simulation. Afterwards, we present our inspiration for Footloose through related work. Finally, we close with Footloose's limitations and changes we plan to make in the future.

3. Footloose Architecture and Data Structures

Footloose has three main responsibilities: storing data, shuttling data between devices, and maintaining data consistency. As such, the implementation breaks into modules and data structures corresponding to these responsibilities. Figure 3 illustrates the breakdown. Applications store data in Footloose by communicating with the Footloose Store (FLS) using `UpdateEvents`, `RecordIds`, and Wishes. Each device in Footloose has an FLS structure containing some of the shared data. The FLS allows Footloose to implement selective conflict resolution. The FLS communicates with the Footloose Protocol Daemon (FPD) to read new data from other devices. The new data is gleaned from other devices during synchronizations during brief connection times. Since physical eventual consistency combines device communication, data routing, and consistency guarantees, these responsibilities are handled wholly by the FPD using `StatusVectors`. Each device runs a FPD.

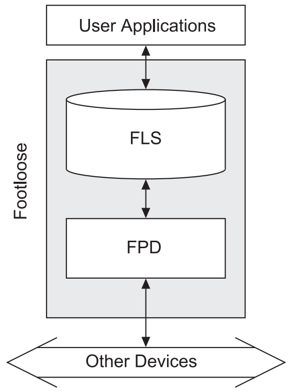

Figure 3. Major components in Footloose.

3.1. The FLS, `RecordId`s, and `UpdateEvent`s

Applications that wish to share data among many devices using Footloose place `UpdateEvents` in the FLS. Each time an `UpdateEvent` is added to the FLS, the FLS increments its `counter` variable. This counter is local to the device and serves only as a measure of age—this "age" is used by the FPD for routing. Each `UpdateEvent` in the FLS represents a change from one specific data state to another—in other words, a `UpdateEvent` is the result of a data write. Applications name specific pieces of data with `RecordIds`. New data is created by updating the `UpdateEvent` associated with a `RecordId` from `null` while it is deleted by updating it to `null`. In the process of creating data, the FLS tags data with an application-specific type. This type is used by the FPD to route data. When applications query the FLS, it only reports back with `UpdateEvents` that have not been updated by other `UpdateEvents`. Hence, from the application's standpoint, the FLS is nothing more than a mutable mapping of `RecordIds` to `UpdateEvents`.

We chose this programming interface because it provides a simple way to allow applications to detect conflicts, to make conflicts opaque to Footloose, and to ensure that no updates are lost. Conflict detection is almost trivial: whenever a `RecordId` maps to more than one valid `UpdateEvent`, there is a conflict. Conflict resolution is therefore similarly easy: the application that reads the `UpdateEvents` in conflict resolves the conflict by determining which `UpdateEvent` is actually correct and updating the `RecordId` to a new `UpdateEvent` representing the "winner" `UpdateEvent`. Note that this is an entirely application-based process since the FLS itself simply stores UpdateEvents without regard to RecordId. Using UpdateEvents rather than standard creates, writes, and deletes ensures that we could identify every change in a uniform way rather than having to peek inside the UpdateEvent to determine changes. These last two properties make the conflict status of UpdateEvents opaque to Footloose itself. Hence, any device can deal with any UpdateEvent in a uniform way, enabling selective conflict resolution.

We realize that the UpdateEvent paradigm may not naturally fit many applications. Therefore, we allow applications to maintain their own data stores and caches outside of Footloose by registering with the FLS. In return, the FLS notifies the application whenever new UpdateEvents have arrived from a synchronization with another device so it can update its own structures. In order to maintain consistency, before a synchronization, the FLS also tells the application when it must reconcile conflicts and flush external data back to Footloose. In the end, instead of continually using UpdateEvents, applications can use external data structures and simply provide a summary of UpdateEvents when Footloose needs them. As a further benefit, this mechanism also gives us a framework for supporting legacy applications. Though we have not yet developed one, a wrapper could sit between the legacy application and the FLS to interpret the FLS's callbacks. It can then update the legacy application using its native API.

3.2. Wishes and Device Interest

Every installed application on a smart device registers with the FLS the types of data that it will put in the FLS and the kinds of data that it wants to read from the FLS. These types match the types associated with the data on creation. We call these registrations *wishes* since they represent the kind of data that the application desires to see. A device that has registrations is *interested* in the data types of its registrations. Footloose uses device interest to make routing decisions, as outlined in the next section. In Ben's phone number scenario, all of the devices are interested in data of type "contact information" so whenever two devices communicate they know to pass along Alyssa's phone number.

Interest registrations live inside the FLS, and consequently are shared among all of the devices. Furthermore, every device registers an interest in the interests of other devices. This means that, as devices communicate, they exchange their knowledge of the interests of other devices using Footloose itself. Over time, as the registrations propagate, each device builds a database of the interests of other devices, which is used by the FPD to route UpdateEvents. This knowledge share enables Louis's scenario. The two end computers have registered their interest in financial data, while the cell phone has no such interest. However, the cell phone has communicated with the two computers, and as such, knows that it can carry financial data to the home computer. The knowledge share also reduces configuration—users merely have to use data and Footloose determines where it should go.

3.3. The FPD and StatusVectors

The FPD is responsible for implementing physical eventual consistency and no lost updates. Like Ficus, it does this using a variation of Parker's version vectors [15] we call StatusVectors. The FPD propagates updates by figuring out which UpdateEvents a partner device needs to see and sending only them. Though this required adding a few more states to the StatusVectors and complicated update algorithms, we chose this over a flood approach since devices have limited storage and communications resources.

The FPD maintains extra metadata in the FLS for every UpdateEvent in a StatusVector. The StatusVector contains a one-byte element for every device in the system, indicating *this* FPD's understanding of the state of this UpdateEvent on every other device. This element may take on one of six values, indicated by single-character codes:

- *: Uninterested. No application on this device has registered a wish for this UpdateEvent
- N: Not seen. This device has not seen this UpdateEvent.
- V: Valid. This device believes this UpdateEvent contains a valid value.
- I: Invalid. This device knows that this UpdateEvent is invalid, having been updated by another UpdateEvent.
- K: Killable. This device knows that all devices know that this UpdateEvent is invalid.
- G: Gone. This device no longer has this UpdateEvent.

The values other than * form a strict ordering from N (least knowledge) to G (most knowledge).

3.3.1. StatusVector Use StatusVectors are used in two different ways by the FPD. First, they indicate whether two devices need to communicate about a given UpdateEvent. Second, they indicate when an UpdateEvent can finally be garbage collected by the FLS.

If device $\mathcal{A}$ is sending events to device $\mathcal{B}$, its FPD will first determine which set $\mathcal{S}$ of devices that $\mathcal{B}$ represents a good route to. We currently use the heuristic that $\mathcal{B}$ is a good route to another device $\mathcal{C}$ if $\mathcal{B}$ saw $\mathcal{C}$ at a later FLS counter number than $\mathcal{A}$ did. $\mathcal{A}$'s FPD searches through its FLS for UpdateEvents where

	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	Event
1	VNN			$\mathcal{D}_1$ creates record
2	VVN	VVN		$\mathcal{D}_1$ synchs with $\mathcal{D}_2$
3	VVN	VVV	VVV	$\mathcal{D}_2$ synchs with $\mathcal{D}_3$
4	VVV	VVV	VVV	$\mathcal{D}_1$ synchs with $\mathcal{D}_2$
5	VVV	VVV	VVI	$\mathcal{D}_3$ updates record
6	VVV	VII	VII	$\mathcal{D}_2$ synchs with $\mathcal{D}_3$
7	KKI	KKI	VII	$\mathcal{D}_1$ synchs with $\mathcal{D}_2$
8	KKI			$\mathcal{D}_2$ synchs with $\mathcal{D}_3$
9				$\mathcal{D}_1$ synchs with $\mathcal{D}_2$

Figure 4. The status vectors stored at three devices over the lifetime of a single `UpdateEvent`.

- some device $\mathcal{D}$ in $\mathcal{S}$ is interested in the `UpdateEvent` and
- $\mathcal{D}$ knows less about the `UpdateEvent` than $\mathcal{A}$.

Only records that match this criteria are sent to $\mathcal{B}$.

As discussed in the Ficus paper, invalid `UpdateEvents` cannot be garbage collected by the system immediately. If they were, it would be impossible to determine whether the absence of an `UpdateEvent` in a given FLS meant that it was invalid, and had been deleted at that node, or valid, but never seen by that FLS. This information is crucial for eventual consistency protocols on devices with limited storage capabilities. Thus, we follow Ficus' example in having a two-phase deletion protocol and can provide the no lost updates guarantee.

Each time a `StatusVector` is updated at an *interested device*, it is advanced using the rules shown below:

1. This device's element is set to the maximum element in the `StatusVector`, showing that this device is now aware of the `UpdateEvent`'s new overall status.
2. If all elements are either `I` or `*`, set the element for this device to `K`.
3. If all elements are either `K` or `*`, purge this element from the FLS.
4. If all elements are either `V` or `*`, and this is a deletion update, set the element for this device to `K`.

Thus, once an FPD knows that all other devices know that an `UpdateEvent` is killable, it can safely purge it. After this point, it will report no knowledge of the `UpdateEvent`. If another device sees that this device has purged the `UpdateEvent`, it may also purge it, since it knows from the `K` element that the `UpdateEvent` was killable.

Figure 4 is an example of these rules in action as applied to a single `UpdateEvent` $\mathcal{U}$. Step 1 is the creation of $\mathcal{U}$ and steps 2-4 show the propagation of the creation. In step 5, $\mathcal{D}_3$ updates the `RecordId` $\mathcal{U}$ represents from $\mathcal{U}$ to $\mathcal{U}'$. This means that at $\mathcal{D}_3$, $\mathcal{U}$ is invalid since $\mathcal{U}'$ has taken its place. Consequently, $\mathcal{D}_3$ marks its view of $\mathcal{U}$ with an `I`. Step 6 shows the propagation of the invalid status. At step 7, $\mathcal{D}_1$ is the last device to be notified of the invalidation. This means that the `UpdateEvent` is considered killable; the status bytes can be advanced to `K`. Finally, steps 8 and 9 show devices purging killable `UpdateEvents`.

These rules do not apply at uninterested devices. A device that is not interested in an `UpdateEvent` may purge it at any time without affecting the correctness of the protocol. The only reason that device would have seen the event is that it is on the path between two interested devices, and the transmitting devices will at some point "re-transmit".

3.3.2. `StatusVector` Updates `StatusVectors` for an `UpdateEvent` are updated by the FLS in the course of standard application writes and by the FPD during synchronization. For every `UpdateEvent` $\mathcal{U}$ received by the FPD of device $\mathcal{A}$, $\mathcal{U}$'s `StatusVector` is merged into $\mathcal{A}$'s FLS. The merged `StatusVector` is computed by taking the pairwise maximum of the `StatusVector` at each device, and advancing according to the rules in section 3.3.1. When the merge process results in modification to the `StatusVector`, the modified `UpdateEvent` is then sent back to the FPD of device $\mathcal{B}$ for merging. This process continues until $\mathcal{A}$ and $\mathcal{B}$ converge on a coherent status of the `StatusVector`. If synchronization is interrupted during this process, the devices revert to their old `StatusVectors`.

3.4. Conflicts and Correctness

Footloose guarantees "no lost updates." The proof that Footloose maintains this is that Footloose will not delete a record until all interested devices have seen that record and marked it as invalid. If the `StatusVector` advancement rules are the only means of purging an `UpdateEvent`, then this property ensures that no updates get lost.

Unfortunately, uninterested devices can purge uninteresting data whenever they need to. However, as long as an update eventually makes it through, such purges only slow down the rate at which updates are propagated. This is again due to the fact that interested devices will not purge an event until it knows that all interested devices have seen the event: one device cannot know the `StatusVector` values of another interested device unless there has been some pathway at some time between it and the other device. Furthermore, since Footloose reverts changes when an `UpdateEvent`'s communication is interrupted, it is not possible for the `UpdateEvent` and its `StatusVectors` to become out of sync. Therefore, if an interested device knows that another device thinks an `UpdateEvent` is killable, we know that this assessment is correct and can rely on the `StatusVector` advancement rules. The first device will keep retransmitting the `UpdateEvent` to the uninterested device since it will still think the uninterested device is a route to the other interested devices until it hears about better routes.

Conflicts are handled easily by Footloose. Recall that a conflict occurs when one `RecordId` maps to more than one valid `UpdateEvent`. Going back to our first scenario, suppose that Alyssa's phone number is stored under `RecordId` $\mathcal{A}$. Originally, all devices agree on a single `UpdateEvent` $\mathcal{U}_A$ for $\mathcal{A}$ containing the value 1 on all of the devices. When she calls with her new number, 2, Ben's cell phone creates an `UpdateEvent` $\mathcal{U}_B$ with value 2 that invalidates $\mathcal{U}_A$. Suppose that Ben leaves his cell phone and PDA at home. At work he gets another call from Alyssa with an even newer phone number, 3, and enters it on his work computer. The work computer creates an `UpdateEvent` $\mathcal{U}_C$ with value 3 that also invalidates $\mathcal{U}_A$. As Ben brings his devices back and forth, they will eventually each get a copy of $\mathcal{U}_B$ and $\mathcal{U}_C$. $\mathcal{U}_A$ will get purged since it is invalid as far as all are concerned. Then suppose that Ben accesses Alyssa's number on his cell phone. The FLS returns two `UpdateEvents` for the `RecordId` $\mathcal{A}$. The cell phone notices this and notifies Ben that there are two possible numbers for Alyssa; in return he chooses that 3 is the correct number. The phone then invalidates both $\mathcal{U}_B$ and $\mathcal{U}_C$ in favor of a new update event $\mathcal{U}_D$ with value 3. As these `UpdateEvents` get shared, $\mathcal{U}_B$ and $\mathcal{U}_C$ will be eventually purged and $\mathcal{U}_D$ will persist.

We realize there are situations where conflicts can persist for a long period of time, for example, if two devices resolve a conflict in opposite directions. However, we do not attempt to fix this: conflict resolution is the sole responsibility of applications, not Footloose.

4. Initial Implementation and Evaluation

We have built an initial implementation of Footloose and a simple phone number application that uses Footloose. Both are written in Java.

4.1. Experience writing an application

Our phone number application has a command-line interface allowing the user to insert name/phone number pairs, and then lookup, edit, and delete the inserted pairs.

The application took only 100 lines of Java code, of which about a quarter deal with the FLS. Of these, only the conflict resolution method (one line) is code that would not need to be written to use this application on top of any other distributed data structure. Footloose behaved as expected when the application was tested, and generated no additional debugging effort.

4.2. Performance Evaluation

4.2.1. Simulation Framework Since Footloose is designed for environments in which disconnection is more the rule than the exception, the rate at which data propagates throughout the network depends much more on the rate at which devices come into proximity with each other than on the base network speed. To make sure that synchronization can be effective even during brief encounters, it is important that the number of messages sent be kept low. Wall-clock time is not important because we expect the data transfered to be small and also because wall clock time is insignificant compared to disconnection time. Therefore, to measure the performance of our protocols, we created a framework that accurately measures the number of synchronizations and messages, while not accurately reflecting the actual amount of wall-clock time required for a given synchronization.

Rather than run over the network, our framework simulates each device with a Java object running in its own thread. A mock link layer emulates wireless connections by maintaining each device's neighborhood and forwarding messages between devices. We chose a simple simulation technique for the order in which synchronizations occur. The devices are assigned to nodes in a connected graph. At each time element, one edge in the graph is selected at random, and the devices assigned to the nodes adjacent to that edge are added to the same neighborhood, synched with each other, and then taken apart again. We do this to simulate a user bouncing among different devices in the worst possible way. In terms of our scenarios, imagine Ben randomly bringing two devices close, letting them synchronize, then bringing them apart, without holding a single device for a long period of time.

We used five different graph topologies: a clique, a ring, a line, a star, and a random tree. In addition to the topology, we adjusted the number of "smart" devices, the number of "dumb" devices, and the number of `UpdateEvents` that fit into each dumb device's FLS. Dumb devices have only Footloose installed. Smart devices also have a single application installed. A dumb device must be connected to at least two other devices: otherwise, it would never participate in message passing.

For each simulation, each smart device was primed with 10 random events (creates, updates, or deletes). The random-synchronization simulation then proceeded until all devices agreed on the state of all events (a consistent state). To ignore overhead from initial system setup of wishes, we then repeated this process, and measured the number of synchronizations and the number of `UpdateEvent` messages (called “updates” in the figures) sent in this second phase. Note that for n smart devices, $10n$ events are generated, which lead to at least $10n^2$ necessary updates in a perfect, fully-connected system. Each data point is the average of five consecutive runs.

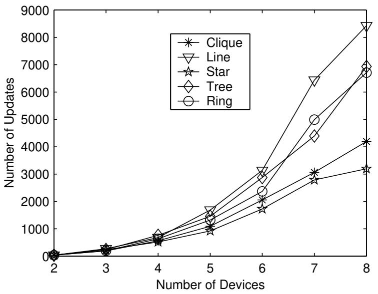

Figure 5. Updates required to reach eventual consistency, by topology and number of devices.

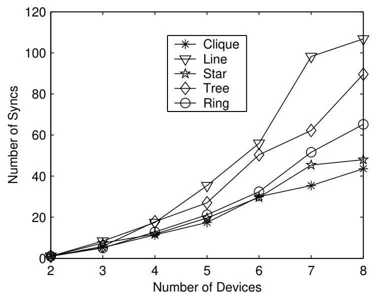

Figure 6. Synchronizations required to reach eventual consistency, by topology and number of devices.

4.2.2. Results Figure 5 shows the results for updates to reach eventual consistency. Notice that, as might be expected, consistency is reached most quickly in the clique and star topologies, and least quickly in the line topology, which contains the longest paths between two nodes. The number of updates for cliques is larger than it might be in a system that takes such strong connectivity as the common case: the routing table stored at each device is a tree, meaning that some messages are sent to third parties that could have been delivered directly. However, the number of updates required for the line topology is only two or three times as much as that required for the clique, even though

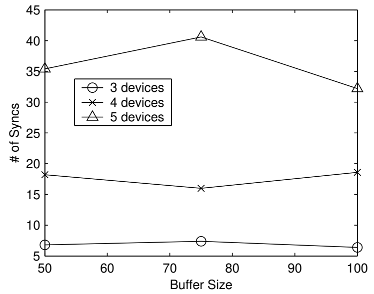

Figure 7. Synchronizations required to reach consistency given limited device capacity.

the paths are five or six times as long. This shows that Footloose’s performance does not vary drastically even on very different network topologies. Therefore, Footloose can be expected to perform well on dynamic topologies created as users roam between devices.

Figure 6 shows the same experiments, this time with number of synchronizations required to reach consistency. This time, the graphs in the best cases (star, tree, and ring) are nearly linear, whereas the line and tree graphs show more susceptibility to the random distribution of our simulation: although it should only require $2n$ synchronizations to bring a line to consistency, this requires these synchronizations to be perfectly ordered from one endpoint down to the other and back, which takes longer to happen randomly as the length of the line gets longer. However, if a user devices can connect to each other as the user moves between them, we believe that a better update order will be easier to come by and Footloose will approach the $2n$ lower bound.

We also evaluated the impact on communication efficiency of the number and capacity of dumb nodes in the network. We placed two smart devices on either end of a line of dumb devices, and varied the number and FLS sizes of the dumb nodes. Figure 7 shows the number of synchronizations required to reach consistency in this system. (The number of devices shown includes the smart endpoints.) The time to consistency depends on the number of dumb nodes, certainly, but is independent of the FLS size of the intervening devices to a point—not shown—where the dumb devices were too small to even hold the wishes of other devices, in which case synchronization failed. Hence, Footloose can work on devices with only a small amount of memory to spare to other applications.

5. Related Work

Perhaps the most relevant work is Roma [18]. Roma is based on transferring and synchronizing metadata to inform the user of the state of her data on all of her devices. Roma enables the user to issue directives saying, for example, "propagate changes," and have the directive be implemented. Roma relies on a centralized database and assumes that the user will always have such a device with her. Updates flow from the database. We make the assumption that users will always have *a* device with them—rather than a specific one–and instead assume that devices can communicate when the user transitions between the two: in other words, we allow update knowledge to flow through the entire device network. This distinction shows up in our consistency models and guarantees. Roma will inform the user where the most recent copy of a piece of data even if it does not have the data on hand. Footloose, in contrast, has no system-wide knowledge of the locations of the most recent pieces of data, but tries to propagate updates (and potential conflicts) as fast as possible to avoid the situation to begin with.

Outside of Roma, we took a back to basics view when designing Footloose, as such we relied heavily on the consistency ideas of the database and distributed filesystems fields. An early review [14] proved useful in comparing many of DFSes. However, Footloose works in an environment where a network connection is a luxury, so we had to discard pessimistic systems like Coda [11] for optimistic ones. Though we did not aim for ACID guarantees, [13] helped us better understand the problems of optimism in a very constrained environment.

Two optimistic file systems, Ficus [5] and Bayou [2], helped shape our thinking as we developed Footloose. Both are designed for connected environments, but allow work during partitions. From Ficus we gained an early implementation of our `StatusVectors`, *one copy availability*, and "no lost updates." Bayou provides application-level conflict resolution: conflicts are resolved by attaching conflict resolution code to every piece of data. In addition to the communication overhead inherent in transporting conflict resolution code, we cannot guarantee that all Footloose devices can actually run their resolution code. Therefore, we developed selective conflict resolution.

There have been several attempts to port DFSes to mobile environments. Rumor [6] most closely matches the goals of Footloose since it keeps the policies of Ficus and allows two peer machines to reconcile differences and gossip updates from other peers. However, in order to reduce the number of update messages, updates are routed in a fixed ring. Footloose, in contrast, works with a dynamic network topology and allows messages to propagate any which way.

Other systems provide for eventual consistency and argue that inconsistency can be tolerated within certain time boundaries. One example is Porcupine [17]. Inconsistency in Porcupine is meant to last for a few seconds, while in Footloose, data can be inconsistent for as long as it takes the user to move between devices. However, this is not a problem. E-mail generates most updates in Porcupine and so Porcupine must work a timescale of seconds or less. Users generate updates in Footloose, so inconsistency can be tolerated as long as inconsistent parts of the system are not in the forefront of the user's experience.

Since Footloose can store arbitrary data types, a comparison to systems like OceanStore [12] or Distributed Data Structures (DDS)[4] is not unwarranted. OceanStore provides a backwards-compatible file-like API and uses strategies like erasure codes, both of which we are considering for future versions of Footloose. Footloose fundamentally differs from both OceanStore and DDS since Footloose distributes data among all participants in the system rather than among a select set of servers.

Group communication systems like Lotus Notes [10] or Horus [21] deal with many of the same issues as Footloose. Horus focuses on application development and attempts to make distributed applications modular, rather than our focus on the user side of the picture. Lotus Notes allows optimism by branching objects into different streams that get recombined later on. Though similar to watching `UpdateEvents`, streams replicate the entire object and are more resource intensive, which would not work in our environment.

Though we chose a variation of version vectors to detect conflicts this was not without reservation. We also considered causal histories. Recently, a new kind of causal history, Hash Histories [9], have been published claiming to cause fewer false conflicts and use space proportional to the number of updates rather than the number of devices. Footloose needs to encode routing information with the `StatusVectors`, so information proportional in size to the number of devices is needed anyway and Hash Histories do not provide savings. We are currently researching dynamic version vectors schemes [16] to allow our implementation to work with dynamically changing device sets.

Finally, some recent research [1, 3, 22] has tried to quantify the amount of inconsistency in optimistic systems. In fact, the papers introduce systems that provide controls that let an application specify how consistent it wants its data. Unfortunately, when consistency bounds are overshot, the systems prevent updates, which would prevent progress in Footloose.

6. Future Work

We have several directions in which we would like to take Footloose. First, we would like to move our implementation to Java-capable handheld computers and cellular telephones. This way we can test real user reactions to Footloose since physical eventual consistency is based on user motion and we do not have good traces of user and device movements.

There are also limitations in our current implementation that we would like to remove. For example, since we cannot dynamically change the number of devices in our `StatusVectors`, we limit the number of devices in the system. We would also like to provide tools for application developers that allow them to create complex data structures out of `UpdateEvents`. These would include a tool to make directory-like structures, which we found we needed as we created more complex applications. Finally, we do not have a comprehensive security framework for Footloose; this needs to be remedied in order to make Footloose useful outside of our small workgroup.

7. Conclusion

Footloose provides a consistent view of user data across mobile devices with disparate computational power. This is accomplished using two techniques. The first, physical eventual consistency, allows us to limit inconsistency by leveraging the fact that users will often have at least one device in their reach at most times. We propagate updates through the communication links formed when two devices come close to each other. Conflicts that arise are reconciled by applications. However, unlike other systems, conflicts can flow through devices that cannot reconcile them to those that can, further increasing the quality of shared data. These contributions are important as the number of mobile devices in our world increases.

References

[1] A. Datta, M. Hauswirth, and K. Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[2] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, Santa Cruz, California, 8-9 1994.

[3] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[4] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction, 2000.

[5] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeir. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer 1990 USENIX Conference*, pages 63–72, 1990.

[6] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER Workshops*, pages 254–265, 1998.

[7] HotSync Synchronization Protocol. http://www.palm.com.

[8] Compaq personal server project. http://crl.research.compaq.com/ projects/ personalserver/.

[9] B. B. Kang, R. Wilensky, and J. Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, 2003.

[10] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozie, and L. Greif. Replicated document management in a group communication system. In *Proceedings of the Conference on Computer Supported Cooperative Work*, 1988.

[11] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[12] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[13] H. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[14] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4), December 1990.

[15] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual consistency in distributed systems. *IEEE Transactions on Software Engineering*, May 1983.

[16] D. Ratner, P. Reiher, and G. J. Popek. Dynamic version vector maintenance. Technical Report CSD-970022, UCLA Department of Computer Science, 1997.

[17] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability, and performance in porcupine: a highly scalable, cluster-based mail service. *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, December 1999.

[18] E. Swierk, E. Kiciman, N. Williams, T. Fukushima, H. Yoshida, V. Laviano, and M. Baker. The roma personal metadata service. *MONET special issue of best papers from WMCSA 2000*, 7(5), October 2002.

[19] SyncML. http://www.syncml.org.

[20] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University.

[21] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4), April 1996.

[22] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 305–318, October 2000.