Footloose: A Distributed Data Buffer for Primarily Disconnected Operation

Justin Mazzola Paluska, David Saff, Tom Yeh, Kathryn Chen

Abstract

In mobile computing environments, users may have many different devices that share the same data sets, but are commonly disconnected from each other. We present a distributed caching and synchronization scheme that provides a shared buffer across such a set of devices using only pair-wise simultaneous communication.

Footloose allows user data to be continually available in spite of disconnection. Moreover, Footloose reduces the amount of data management that the user must manually do. A typical application of Footloose would be maintaining a user's contact information across a cell phone, PDA, and desktop machine. Footloose allows the user to simply modify data on any three of the devices and have it eventually be consistent among the set.

Footloose guarantees eventual consistency and no lost updates. It is not necessary for there to ever be communication between any given pair of nodes, as long as the communication pathways form a connected graph over time. Different devices can register interest in different data types, and devices interested in a given data type need not form a connected graph. A conflict is allowed to persist until it reaches an application that can resolve it. Communication overhead scales reasonably and polynomially with the number of participating devices.

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, an explosion of personal peripherals, including PDAs, digital cameras, cell phones, MP3 players, laptop and palmtop computers, wireless pagers, and more, have given the PC the additional role of a "base station" or "orchestra conductor". Each of these devices is synchronized, or "synched" regularly with the "Home PC". The Home PC is responsible for making sure that the data from a peripheral is persisted. Data from the peripheral might also possibly made available for new applications; for example, pictures from a camera might now be accessible to be put into a slideshow. The Home PC also sends data to each peripheral, which must be up-to-date and relevant to the device's use patterns; for example, a cell phone might only need names and phone numbers. To use another metaphor, the stationary home computer serves as the central nervous system for a distributed collection of devices, each of which is in regular contact with the home computer. This central nervous system provides *consistency*, a guarantee that, to the extent possible, interacting with one of these devices is equivalent to interacting with the Home PC.

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 their 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, 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 implementation of Footloose, a distributed, disconnected system designed to allow users and distributed applications to maintain a personal knowledge base. As long as a personal network of devices is never completely partitioned in space-time, we guarantee *eventual consistency*, even if none of the devices running a particular application talk directly with each other.

2 Footloose: the concept

Footloose is a distributed, disconnected buffer system that allows applications scattered across many devices to share data. Footloose provides an integrated framework for data storage, data routing, and data transfer. Applications are presented with a simple API through which they can insert data into and extract data from the system and register interest in data from remote devices. The Footloose system ensures that all other interested components in the system see changes that registered applications make to shared data. Furthermore, any data stored by applications in Footloose will be protected by a few guarantees discussed later.

The goals of Footloose are not novel. Historically, replicated file systems attempt to achieve much the same goals. In fact, the design of Footloose is heavily influenced by two replicated file systems: Bayou [DPS+94] and Ficus [GHM+90]. Both systems allow *optimistic* operation in the face of disconnection or network partition. However, they differ on their approach to conflict resolution. Ficus will only resolve conflicts if there is a *resolver* installed on the system. If not, the conflict is thrown up to the user to manually resolve. Bayou embeds arbitrary snippets of code in the data itself that should be run to resolve conflicts. Neither approach is suitable for the typical environment in which we expect Footloose to operation.

We settled on a different kind of design after making a few assumptions about the environment in which Footloose will live:

- Connections between devices are fleeting but do form a connected graph over time. However, the set of devices on which any given application runs need not form a connected graph over time.
- Not every device has the resources to be able to resolve conflicts.
- The set of devices in the system does not change.
- Devices have only a finite amount of storage.

Since Footloose works primarily in a primarily disconnected network environment, it is obliged to allow optimistic updates. What makes Footloose different from the above systems is the way it handles conflicts. Like Ficus, Footloose provides application-level conflict resolution on devices that have the application installed. However, it does not require every device to run every application or application resolver. Rather Footloose allows *dumb* devices—those that do not have resolvers—to carry potentially conflicting data to *smart* devices—those that have applications installed and can resolve conflicts. Conflicts flow *through* dumb devices until they reach a smart device that can resolve the conflict. The resolved conflicts then flow back through the dumb devices to interested applications. Different devices may have different applications installed, so what appears to be a dumb device to one application may be a smart device to another.

Dumb devices in our system are analogous to data packets on the Internet, with the applications as Internet hosts. Dumb devices merely act as a way to transfer data between interested endpoints.

Footloose provides two guarantees on data to applications: eventual consistency and “no lost updates.” Eventual consistency means that, while at any given point devices may have conflicting views on the state of the system, given enough synchronizations, all of the devices will agree on one state. “No lost updates” is borrowed from Ficus—it means that any update in the system will be seen by any device interested in the update. Since devices have finite storage, dumb devices can purge their buffers when necessary. However,

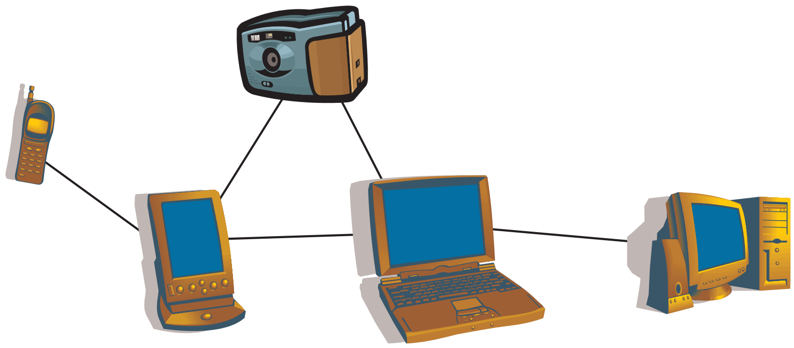


Figure 1: A typical set of devices and communication pathways

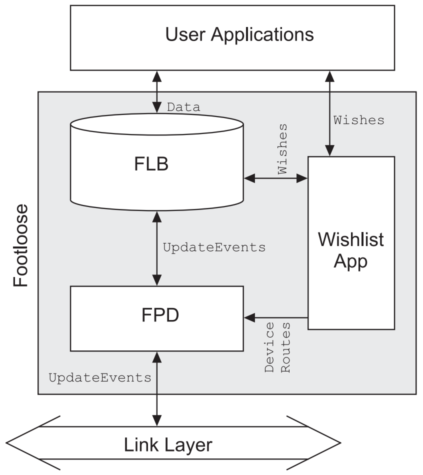


Figure 2: Modules on a single device in Footloose

`Type` — tagged data types
- *appid* — application identifier
- *filterid* — application data type identifier

`RecordId` — tags records
- *type* — `Type` of this
- *recordid* — application-level record identifier

`BufferId` — tags instances of records
- *frid* — the record of which this is an instance
- *version* — system unique version

`UpdateEvent` — data change record
- *id* — `BufferId` uniquely identifying this
- *replacedVersion* — the `UpdateEvent` updated by this
- *data* — the actual data change
- *dirty* — dirty bit

Figure 3: Major data structures in Footloose

keeping with our analogy, we design our algorithms so that purges will cause no more harm than lost TCP packets—given retries, the data can get through eventually, so our system will make progress.

3 Footloose: Architecture and Implementation

3.1 Overview of Modules and Data Structures

The major components and data structures of Footloose are summarized in Figures 2 and 3.

Applications that want to use Footloose to manage their shared state must be programmed around `UpdateEvents`. Abstractly, an `UpdateEvent` represents a unique change from a specific data state to a new data state. A new piece of data is created by updating the data state `null` while a piece of data is deleted by updating it to `null`. Applications identify their data using a `RecordId`. Importantly, a single `RecordId` can map to multiple `UpdateEvents`; this indicates a conflict. Applications then can resolve the conflict however they choose.

Footloose consists of three components: the Footloose Buffer (FLB), the WishlistApplication, and the Footloose Protocol Daemon (FPD). Each is responsible for one part of ensuring that the right `UpdateEvents` get sent to the correct devices. The FLB stores `UpdateEvents`. The FLBs of dumb devices can participate in passing data since they must only move `UpdateEvents`—whether or not the `UpdateEvents` result in a conflict is opaque to dumb devices. The WishlistApplication on each device determines what `UpdateEvents` other devices should have. This is done by registering *wishes* for data of a specific `Type` with the system. Whenever two devices communicate, they share their wishes. Over time, each device in the system will be aware of what `Types` of data interest other devices and will be able to make routing decisions. Finally, the FPD performs synchronization between two devices and is responsible for transmitting data.

3.2 The FLB

The FLB—or Footloose Buffer—manages the storage of data in the Footloose system, providing a mapping between `RecordIds` and sets of `UpdateEvents`. The FLB exports an `UpdateEvent`-centric API, shown in Figure 4 that controls the visibility of data to applications. `UpdateEvents` themselves are managed by the FLB.

3.2.1 `UpdateEvents`

In order for data associated with deletions and modifications to reach all interested parties, Footloose *updates* data rather deleting or directly modifying it. For example, if a Device $\mathcal{D}$'s FLB were to directly delete data, then when $\mathcal{D}$ synchronizes with other devices, the other devices would not know whether $\mathcal{D}$ does not have a record because it has not heard of it yet or if $\mathcal{D}$ had deleted the record. Similarly, if the FLB were to directly modify a data record, then it would be difficult to determine whether two records conflict or whether one

`createRecord(RecordId frId, data)` returns `BufferId`
  Create a new piece of data
`deleteRecord(BufferId c)` returns `BufferId`
  Delete an existing piece of data
`updateRecord(RecordId frid, newVal)` returns `BufferId`
  Replace the content of an existing data record with `newVal`
`resolveConflict(UpdateEvent replaced, UpdateEvent replacing)`
  Resolve two conflicting `UpdateEvents`, `replaced` and `replacing`, to the winning `UpdateEvent`, `replacing`.
`getRecords(Type t)` returns `List<UpdateEvent>`
  Get the set of `UpdateEvents` with `Typet`.
`getRecords(RecordId frId)` returns `List<UpdateEvent>`
  Get the set of `UpdateEvents` with `RecordId frId`
`getDirty()` returns `List<UpdateEvent>`
  Get the set of `UpdateEvents` which are marked *dirty*
`setRecordProperties(BufferId c, CacheProperties p)` returns `CacheProperties`
  sets the properties of the `UpdateEvent`identified by `c` to properties `p`

Figure 4: API exported by the FLB to applications

replaces the other. Thus, the FLB creates new `UpdateEvents` to replace the `UpdateEvents` representing the data record being deleted or updated. The replaced `UpdateEvent` is no longer be visible to the application, but will exist in the FLB. Replaced `UpdateEvents` are used by the FPD during synchronization, discussed in section 3.4.2.

3.2.2 Resolving Conflicts

During synchronization, data records that a device is interested in are inserted into its FLB. These records, inserted from a synchronization, are marked as *dirty* (see section 3.5 for more details). Applications use `getDirty` to retrieve dirty records. The application is responsible for determining whether the insertions resulted in conflicts and resolving any conflicts that may have arisen from the newly inserted data. To aid applications, the FLB provides the method `resolveConflict`—which allows informs the FLB of which record in a conflict "wins" and which must be invalidated. After the processing dirty data, the application can use `setProperties` to clear the dirty bit of the records it processed.

3.3 The WishlistApplication

The WishlistApplication provides routing data for the FPD by maintaining *wishlists* and the *Visits* set. A wishlist is a set of *wishes*—data types that a device would like to keep updated in its FLB. A device is *interested* in a data type if it has an application installed that uses that data type. A visit is a measure of how recently the current device has communicated with a remote device. Concretely, a visit is a tuple of a remote device and the device's ascending counter. The device increments its counter whenever there is a change to the FLB.

The Visits set allows the FPD to determine to what devices its partner is likely to pass data, while wishlists allow the the FPD to determine what kind of data would be most useful to pass to its partner. Every time a device synchronizes with a partner $\mathcal{P}$, it notes any changes in the partner's wishlists and Visits set.

The WishlistApplication exports five functions as shown in Figure 5. The first three functions allow a device to change its intrinsic wishes. Typically, these will only be called when an application is installed or removed. The last two functions return data the FLB can use for making routing decisions.

3.3.1 WishlistRecords

Wishlists are necessarily shared by all WishlistApplications in the system so that `UpdateEvents` can be forwarded properly. Therefore, each WishlistApplication keeps its wishes in the FLB, allowing the FPD to make all other devices aware of the changes. Furthermore, each WishlistApplication keeps at least one wish—one for other wishlists. This is so that it is kept up-to-date with other devices' wishlists.

`addWish(appid a, filterid f)`
  Add a new intrinsic wish for data of `Type(a, f)`
`removeWish(appid a)`
  Remove all intrinsic wishes associated with Application `a`
`removeWish(appid a, filterid f)`
  Remove the intrinsic wish for data of `Type(a, f)` if one exists
`represents(Device d, Set<Visit> s)` returns `Set<Device>`
  return the set of Devices that `d` provides a route to given its last visit times `s`.
`getTypes(Set<Device> s)` returns `Set<Type>`
  return the set of types that the devices in `s` are interested in

Figure 5: API provided by the WishlistApplication.

Concretely, each element in the wishlist is stored as a `WishlistRecord` in the FLB. A `WishlistRecord` is a tuple of a `Type` and the device originating the wish. The `Type` denotes the kind of data the wish covers. The originating device field allows the WishlistApplication to determine which devices are interested in what.

If there is conflict between `WishlistRecord`s in the FLB, the conflict is resolved conservatively by assuming devices are interested in data rather than not.

The Routes Map and Visits Each WishlistApplication keeps its own unshared Visits set $\mathcal{V}$. When the FPD calls the `represents(Device d, Set<Visit> s)` function, the WishlistApplication scans through both its visit set $\mathcal{S}$ and `d`'s visit set `s`. If `s` contains a visit `v` to remote device $\mathcal{D}$ that is more recent than $\mathcal{S}$'s visit $\mathcal{V}$ to $\mathcal{D}$, then `d` is a route to $\mathcal{D}$. Therefore, the $\mathcal{D}$ is added to the set of devices that `d` represents. Then, $\mathcal{V}$ is updated to contain the count of `v`-1. Hence, a visit for a device is always the most recent counter known if two devices have communicated directly or a slightly smaller value if they have not. This way, the WishlistApplication can determine which device, of a set of devices, has synchronized most recently with a remote device. We use the heuristic that the device that has talked most recently to remote device is the best route to that device. After both sets have been compared, the WishlistApplication updates its visit counter for `d` to be `d`'s current visit counter, indicating it has recently communicated with `d`.

3.4 The FPD

The FPD—or Footloose Protocol Daemon—is responsible for backing up Footloose's guarantees of eventual consistency and no lost updates, without flooding the network or innocent devices with useless messages. It does this by passing messages between Footloose devices, communicating with the local FLB, WishlistApplication, and other applications. Below, we briefly describe the interface between the FPD and lower network layers. We then overview the protocol followed by the FPDs on two devices when they synchronize. Then, we look at extra metadata stored by the FPD in the FLB to keep track of the progress toward consistency. Finally, we look at how the FPD interacts with individual applications on a device.

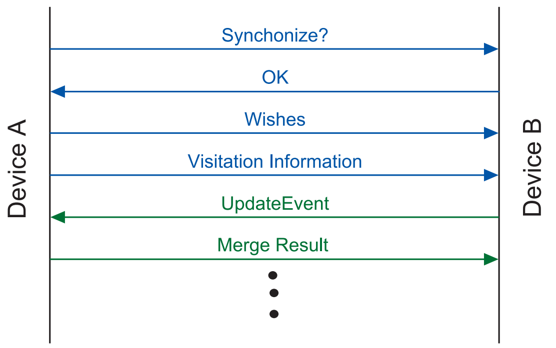


Figure 6: Messages passed during synchronization of device $\mathcal{A}$ with device $\mathcal{B}$.

$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	Event
VNN			$\mathcal{D}_1$ creates record
VVN	VVN		$\mathcal{D}_1$ synchs with $\mathcal{D}_2$
VVN	VVV	VVV	$\mathcal{D}_2$ synchs with $\mathcal{D}_3$
VVV	VVV	VVV	$\mathcal{D}_1$ synchs with $\mathcal{D}_2$
VVV	VVV	VVI	$\mathcal{D}_3$ updates record
VVV	VII	VII	$\mathcal{D}_2$ synchs with $\mathcal{D}_3$
KKI	KKI	VII	$\mathcal{D}_1$ synchs with $\mathcal{D}_2$
KKI			$\mathcal{D}_2$ synchs with $\mathcal{D}_3$
			$\mathcal{D}_1$ synchs with $\mathcal{D}_2$

Figure 7: The status vectors stored at three devices over the lifetime of a single event.

3.4.1 The LinkLayer

The FPD works with the LinkLayer, which is a wrapper for a connection-based network protocol handler, and is discussed only briefly here. For each device $\mathcal{D}$, the LinkLayer maintains a local neighborhood, which is the set of other devices with which $\mathcal{D}$ can currently communicate. The LinkLayer is responsible for notifying the FPD on a device when another device enters or leaves its neighborhood. Based on this information, the FPD uses a simple periodic round-robin selection strategy to choose each device in its neighborhood to synchronize with.

3.4.2 Pairwise Synchronization

Figure 6 shows the sequence of actions taken during one phase of the synchronization. First, $\mathcal{A}$ and $\mathcal{B}$ exchange a handshake `Synchronize?` message to ensure both devices are ready to begin. $\mathcal{A}$'s FPD then instructs the WishlistApplication to clean itself to ensure consistency. (More information on cleaning is in section 3.5). $\mathcal{A}$ then sends all of its `WishlistRecords` and `Visits`.

Upon receiving data, the FPD of device $\mathcal{B}$ will insert into $\mathcal{B}$'s FLB the wishes from device $\mathcal{A}$ and notify $\mathcal{B}$'s WishlistApplication. The WishlistApplication then cleans itself by incorporating these new changes. Then $\mathcal{B}$ calls the WishlistApplication's `represents` function to determine the set of other devices that device $\mathcal{A}$ represents and compiles a list of all the wishes pertinent to these devices.

Using such information, $\mathcal{B}$'s FPD asks each relevant application to clean itself and then, using the metadata described in section 3.4.3, generates a list of `UpdateEvents` useful to device $\mathcal{A}$ or the devices $\mathcal{A}$ represents. Each event in this list is then sent to $\mathcal{A}$, and $\mathcal{A}$ and $\mathcal{B}$ may exchange further messages until they are in agreement on that event's state.

After all the `UpdateEvents` are processed, this direction of synchronization will be considered complete. The direction is then reversed to complete synchronization.

3.4.3 Metadata for Enforcing Guarantees: `StatusVectors`

One way to guarantee eventual consistency is for every FPD to periodically flood its neighborhood with every `UpdateEvent` it knows of. This has several problems. First, it is inefficient with respect to network resources. Second, the only communication between two neighborhoods may be a dumb device with limited memory, and that memory may be wasted on obsolete or uninteresting messages.

`StatusVector` Contents Instead, the FPD maintains extra metadata in the FLB 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 each other device. This element may take on one of six values, indicated by single-character codes:

- `*`: Uninterested. This `UpdateEvent` is not on this device's wishlist
- `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 replaced by a more recent `UpdateEvent`.
- `K`: Killable. This device knows that all devices know that this `UpdateEvent` is invalid
- `G`: Gone. This device no longer is caching this event.

A status vector can be represented by a string of these characters, ordered by device ids. For example, imagine devices $\mathcal{D}_1$ and $\mathcal{D}_2$ believe a `UpdateEvent` is valid, but $\mathcal{D}_3$ has not seen it yet. If $\mathcal{D}_1$ knows all these facts, the `StatusVector` stored for this `UpdateEvent` there will be `VVN`. The values other than `*` form a strict ordering from `N` (least knowledge) to `G` (most knowledge).

`StatusVector` Updating A `StatusVector` is updated in one of two ways. First, as the FLB receives new creations, updates, and deletions, it adjusts the `StatusVector` for each `UpdateEvent` to reflect the current status at this device.

Second, each `UpdateEvent` u received by the FPD of device $\mathcal{A}$ is merged into $\mathcal{A}$'s FLB. The merged `StatusVector` is computed by taking the pairwise maximum of the `StatusVector` at each device, and advancing according to the rules in section 3.4.3. This can result in a modification to the status of u. The modified `UpdateEvent` is then sent back to the FPD of device $\mathcal{B}$ for merging. If there is any change in status of the u as the result of merging on $\mathcal{B}$, u is sent back to device $\mathcal{A}$. This process continues until device $\mathcal{A}$ and device $\mathcal{B}$ can agree upon a coherent status of u.

`StatusVector` Use `StatusVector`s 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 removed from the FLB.

If device $\mathcal{A}$ is sending events to device $\mathcal{B}$, its FPD will first ask $\mathcal{A}$'s WishlistApplication for the set R of devices represented by $\mathcal{B}$. The set of records to send, then, is filtered both by the wishlists of these applications and by only sending those events about which some device in R knows less than $\mathcal{A}$ does, using the ordering on the `StatusVector` elements.

As discussed in the Ficus paper, invalid events can not simply be dropped from the system immediately. If they were, it would be impossible to determine whether the absence of an event in a given FLB meant that it was invalid, and had been deleted at that node, or valid, and never seen at that node. Thus, we follow Ficus' example in having a two-phase deletion protocol and therefore provide the no lost updates guarantee. Each time a `StatusVector` is updated at an interested device, it is advanced using the rules shown in the following section.

`StatusVector` Advancement Rules

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 FLB.
4. If all elements are either `V` or `*`, and this is a deletion event, 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. For an example of these rules in action, see Figure 7.

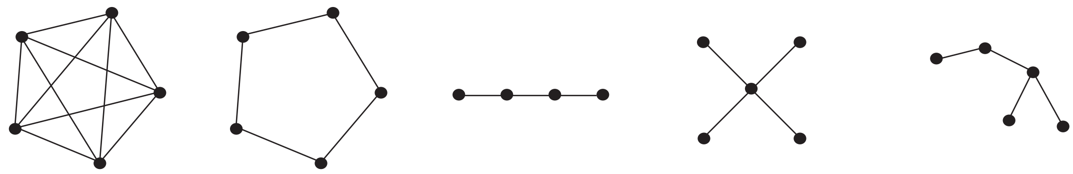
Notice that these rules do not apply at uninterested devices. A device that is not interested in an event may purge it at any time without affecting the correctness of the protocol. The only reason that device would have seen the event is that it is on the path between two interested devices, and the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

3.5 Dirty and Clean Data

Data in the FLB may be updated by either applications or the FPD. It would be inefficient for the application to always be forced to keep the FLB completely up to date: some applications may have internal caches or data structures that do not map well to the FLB's structure. Also, it would be inefficient for the application to always have to check the FLB for updates. To solve these problems, the FPD communicates with an application before and after synchronizing any of the application's data. As shown in section 3.2.2, this mechanism also helps support conflict resolution.

Before synchronization, the FPD calls the application's `clean` method, and blocks until it returns. This signals the application to make sure that the FLB accurately reflects the application's current state, and to handle any dirty records that may be left over from a previous synchronization. During synchronization, any new events inserted are marked dirty.

After synchronization, the FPD issues a non-blocking call to the application's `nowDirty` method. This signals the application that synchronization may have added records to the FLB, allowing the application



(a) Clique (b) Ring (c) Line (d) Star (e) Random Tree

Figure 8: Topologies used to gather scaling data

to resolve conflicts and incorporate this data into any internal structures. Depending on how resilient the application is to volatile data, the application may block between calls to `clean` and `nowDirty`.

4 Evaluation

Here, we present our experiences building a small application on top of Footloose, and show how the network traffic generated by Footloose changes based on a number of parameters.

4.1 Experience writing an application

To test Footloose as a platform for applications, we created a small phonebook application. This 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 FLB. 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. 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 LinkLayer maintains each device's neighborhood, and forwards 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 used five different graph topologies, shown in Figure 8. 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 FLB. Dumb devices have only the WishlistApplication 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

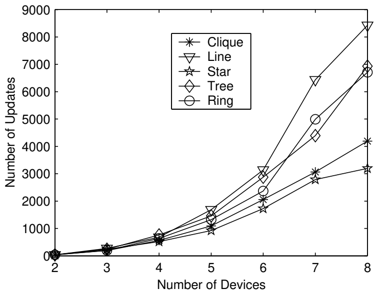


Figure 9: Updates required to reach eventual consistency, by topology and number of devices.

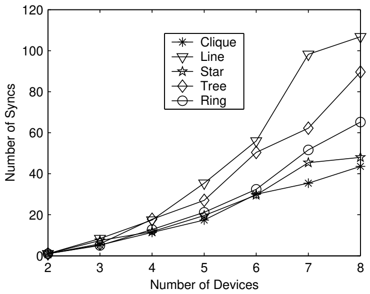


Figure 10: Synchronizations required to reach eventual consistency, by topology and number of devices.

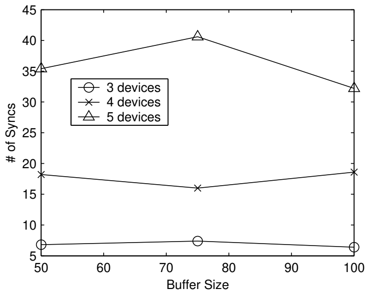


Figure 11: Synchronizations required to reach consistency given limited device capacity.

consistent state). To ignore overhead from wishlist stabilization, 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.

4.2.2 Results

Figure 9 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 the paths in that case are two or three times as long. This shows that Footloose's performance does not vary drastically even on very different network topologies. The number of updates grows as approximately the square of the number of devices, as predicted.

Figure 10 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.

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 cache sizes of the dumb nodes. Figure 11 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 size of the intervening devices to a point—not shown—where the dumb devices were too small to even hold the wishlist data, in which case synchronization failed.

5 Related Work

Many systems exist for either the synchronization or migration of data. For example, protocols such as Palm's HotSync [Pal] or the SyncML Initiative's SyncML [Syn] allow two devices that share an application to synchronize their data. However, both protocols are made for point-to-point synchronization between two devices in contrast to Footloose's synchronization through third-party devices.

In some ways, Footloose resembles remote or disconnected filesystems. However, many disconnected filesystems use pessimistic locking of files. One such system is Coda [KS91]. Obviously, with multiple devices concurrently working on data, pessimistic locking would not work. Therefore, Footloose builds upon *one-copy availability* as introduced in Ficus [GHM+90] to provide for optimistic availability. Footloose differs from Ficus in its emphasis on having a device take an interest in whether another device has received its update. This allows Ficus to facilitate sharing through uninterested devices, which Ficus cannot allow.

Xerox PARC's Bayou project [DPS+94] influenced our decision to include application-level conflict resolution rather than forcing a specific method of conflict resolution. Bayou performs application-level resolution by attaching executable code to every piece of data. Unfortunately, our anticipated environment may not have the bandwidth to continually transport code with records nor the computational power to run arbitrary code, so we adapt our system that only devices with an installed application can do conflict resolution for that application.

Finally, there is some similarity between the map structure of FLB and the distributed hash table of the the Distributed Data Structures (DDS) system [GBHC00]. However the two differ in important ways. For one, the FLB allows a *set* of events to be mapped by a key. Secondly, individual FLBs are not distributed but rather the data that all FLBs hold is.

6 Conclusion and Possible Improvements

Footloose does guarantee the promised properties of eventual consistency and no lost updates, and does so with communication traffic that scales reasonably with the number of devices and length of communication paths in the system. However, the protocol could be optimized in various ways. For example, if there are several equally long paths between two devices, events will travel along both, since it is not clear which is faster. If the total network communication cost was more important than quickly reaching consistency, the protocol could be adjusted to only try one path at a time.

Also, if one device is down or partitioned from the network a long time, the FLBs of other devices may become full with events that the failed device still needs to see. There are ways to collapse a chain of update events to save storage in such situations, which could be added to Footloose.

Finally, there are additional services that might be appreciated by application writers. One example is a standard directory application. Another would be the possibility of generating data that you do not intend to store locally: an example would be a digital camera, which does not need to store pictures once they have been archived on a PC.

References

[DPS+94] 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.

[GBHC00] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction, 2000.

[GHM+90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeir. Implementation of the Ficus Replicated File System. pages 63–72, Summer 1990.

[KS91] 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.

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

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