The syncing algorithm constitutes the core of our notification system. The security protocol protects against infinite loops, unpredictable behavior (race conditions), and inefficiencies. The challenge of syncing is multi-fold, but the statement of the problem is thus:

1. Local data should be backed up on the cloud.
2. Devices should be able to add events to channels (broadcasting role), and devices should be able to receive those events (subscribing role).
3. The device may be offline (turned off, lost battery, poor signal) in which case changes should be queued on the device and broadcasts queued on the server.
4. The device may go offline during a communication, in which case the server and device should be able to complete the communication without issue once a network connection is reestablished.
5. Cross-listing availabilities of multiple users should keep each user’s schedule secure.
6. We assume that the server-database communication is strong and secure.

We respond to the five issues as they are enumerated above (1-5).

**Issue One**

We introduce the private data protocol. When a user makes changes to their private schedule, a communication begins between the server and user’s device. These changes are sent to the cloud, and data is sent back to confirm the changes and look for more changes. If more changes have been made, the new changes are sent to the cloud, and the server responds again to confirm the local and cloud schedules are synced. This process repeats until the server responds and no changes are remaining on the device. In one model, every change is immediately sent to the cloud. This presents the issue of a race condition- one change may create an event, and another may delete it. If the second change reaches the server first, the second change will error, or the first change will create the event and it will never be deleted. Furthermore, even if they reach the server in the correct order, the server cannot confirm the schedule on the database with the schedule on the device, because other changes may have been made on the device and may be on their way to the server.

We propose a model in which the first change is sent immediately, and any further changes are collected on the front-end and not sent until the server responds. Through this, changes will not be received out of order but queued on the device. In addition to solving the race condition, less back-and-forth communication is required if changes are being sent in groups, and furthermore changes (such as creating an event, and then deleting it) are compiled and will cancel, increasing efficiency. Finally, each time the server responds it can check against the schedule on the device, and the communication will cease once the a server confirmation matches the local schedule and the device marks itself as synced.

**Issue Two**

We introduce the broadcasting protocol. A user makes a change to his public channel, and that change is sent to the server in the manner described above. Then all phones are notified of this change. This protocol is functionally similar to the private data protocol- the event changes hit the server, the server confirms, and a communication loop begins if there are new changes on the front-end. Unlike the previous protocol, this change is then to be broadcasted to all instances of the app that are subscribed to the channel. A notification is created on every reachable device. Because not all devices may be online, the server also logs this change for when devices come online. This is detailed below.

**Issue Three**

We introduce the offline protocol. If a device has made changes and it cannot reach the server, the device will collect its changes in a pending list. Once connection is reestablished, it will resend the pending list. Secondly, if a change is broadcasted to the server, the changes will only immediately reach and notify devices which are online. So the server will also log the change to be broadcasted, and offline devices which come online will ask the server to check the log for any channel broadcasts since the time of the last sync.

We raise and address another issue: both the server and device have changes they would like to share. The private data on the device and new broadcasts on the server can be synced independently. But RSVPs on the device and new broadcasts are not independent- perhaps the user has said that they will attend a party, and that party has changed times. In one model we may treat these independently, and the device is notified of the time change, but the device may already have notified the server that they are attending. To prevent this, we may have a correction factor- the device is notified of the time change, and now signals to the server that its decision should be ignored. We fall into another race condition: this signal may reach the server too soon and the RSVP will come second and say the user is going, or the user may indeed wish to go and RSVP to the new time, and the signal will overwrite the old RSVP as well as the new RSVP. Instead of the correction factor, we may instead regard each event’s serial ID, and editing an event will change the ID, so once the RSVP reaches the server, it finds that the old event no longer exists, and it dies. But a new event may assume that old ID unless a history is maintained. And if editing the event changes the ID, changing the party time will notify subscribers that the event has been deleted and a new event with all the same logistics but a different time has been created. This is confusing to users and duplicates unnecessary information in the logs on the server, in addition to the history of used serial IDs that has to be maintained. To prevent the confusion and consolidate the old and new event, overly complicated algorithms could be implemented to compare the events and find if they are similar enough, then the “creation” and “deletion” should be combined into one “edit” notification. This is all highly undesirable.

We propose first receiving the server’s information, incorporating this information on the front-end of the app, and then sending the compounded information to the server. So if the party has been edited, this information will reach the device, and delete the user’s decision to attend the party at the overwritten time.

**Issue Four**

This issue is addressed already by the three protocols above. The device and server do not mark themselves as synced until the communication has completed entirely and a final communication confirms that there are no further changes to resolve and the local and cloud schedules match. Thus if communication falters at any point in the process, the device and server will restart the syncing process.

**Issue Five**

This is addressed by never sending one user’s schedule to another user’s device. If a team member wishes to set up a meeting, the schedules of his team members will not be sent to his device to cross-list availabilities. A clever cracker could view this data by causing it to persist, or mimicking a request to the server for it, or intercepting it, posing a serious security threat. Instead, the server serves as a middleman and handles the cross-listing. The team member simply sends a request to the server, and only the server accesses the team members’ data.

Either way, this cross-listing function requires that schedules be in the cloud, because the devices of the other team members could be offline. This provides stronger motivation for syncing schedules between the server and devices, in addition to simply backing up user schedules.

**Conclusion and Proof of Concept**

By the above reasoning and justifications, we have architected a light communication protocol. In theory it meets the above requirements and therefore affords the following securities:

* Devices behave seamlessly if they go or are offline.
* The cross-listing function occurs on the server instead of on any individual device.
* The server confirms the success of every data transmission.
* The edge case where data transmissions are handled by the server out of order.
* The edge case where a second data transmission is sent out before the server can confirm the success of first data transmission.

As a proof of concept, we implement and simulate this communication framework as a programmed set of Java classes. Devices are randomly connected and disconnected from the server and given simulated input events, but if the user’s events reach the server in order, without repetition, and in their entirety, without crashing, at a reasonable speed, and terminating successfully, we have successfully proven its robustness.