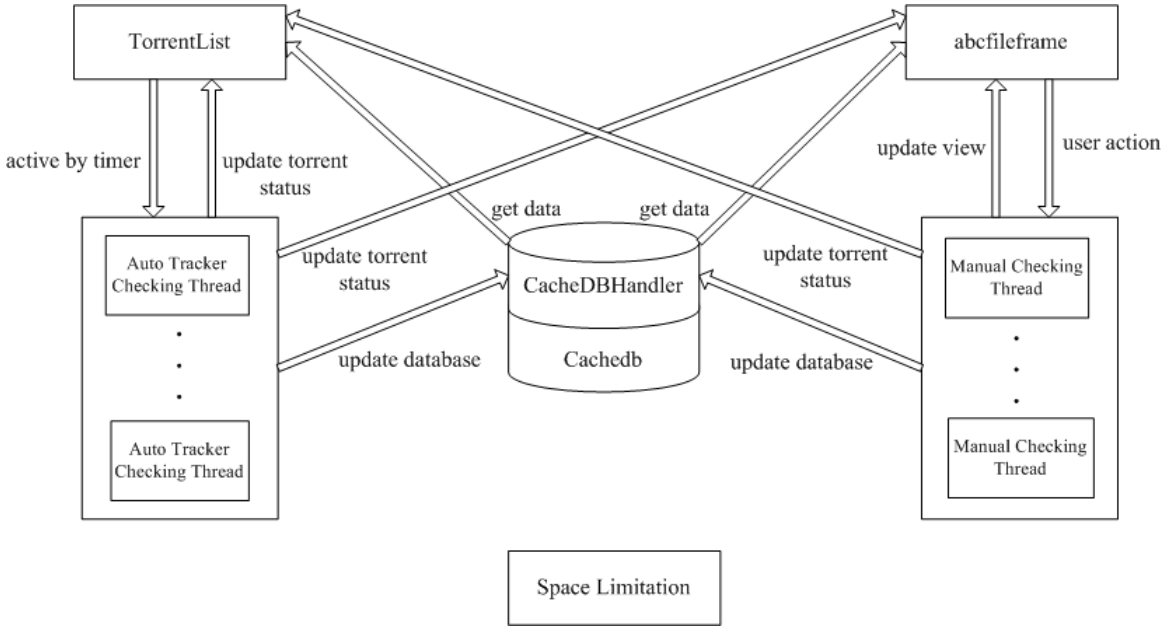

Figure 1: old

Document:	**Tribler Design Note**
Subject:	**Tracker Checking Detail Design with Observer Pattern**
Author:	**Yuan Yuan**
Date:	**August 11, 2006**
Status:	*DRAFT*

1 Overview

Implementing a background module ”auto tracker checking” to check the status of torrent files and offering the user the ability ”manual tracker checking” to check the status of some torrent files in a short time.

2 Old Design

In the old design. The ”checking thread” created by either ”auto tracker checking” or ”manual tracker checking” must know the existence of other modules which also use the same torrent data. When the ”checking thread” find a new status of a certain torrent. First it will update the torrent status in database, then it will update the torrent status in other modules.

Since there are more than one threads which will operate the torrent data in each module, so we must deal with the synchronization problem. The simplest method is to add a Lock in each module. Each time when a thread want to operate the torrent data in module, it must first obtain the Lock of this module.

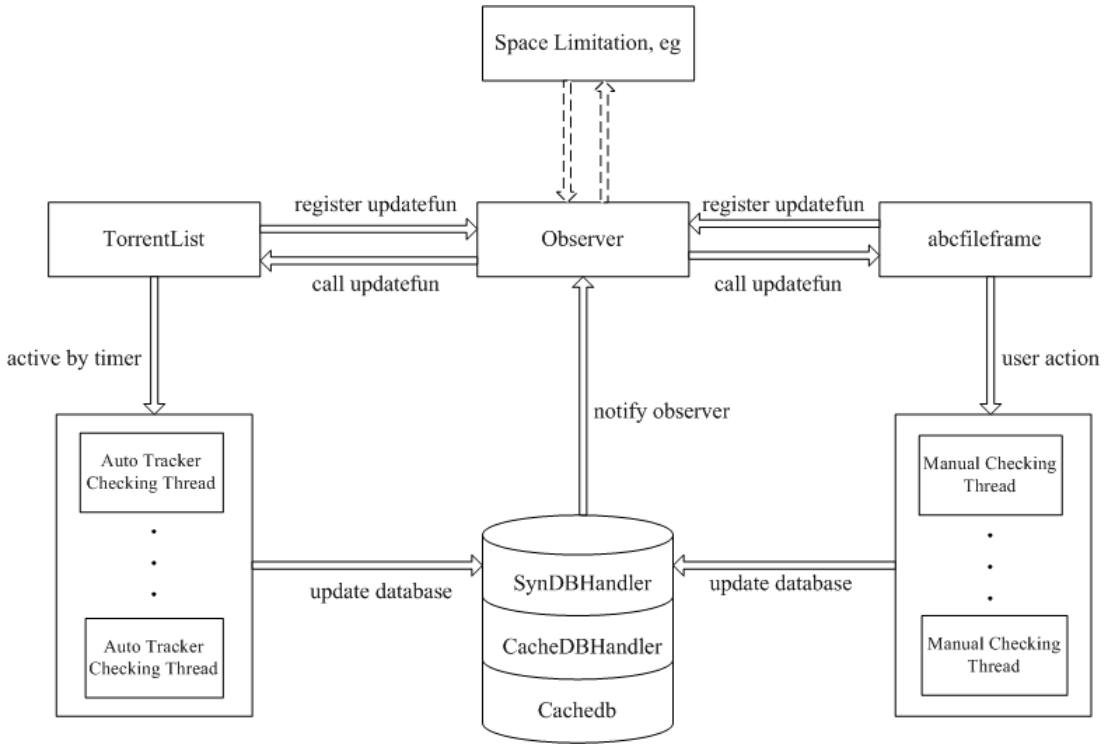

Figure 2: observer pattern

This design is simple, and indeed work. But also have some drawbacks, which limit the scalability of the program. First, each ”checking thread” must exactly know which module has to be updated. This will cause too many connections between modules. For example, if the “checking thread” want to change the torrent data in the FileList Dialog, at first it must know whether this Dialog is created or not. Second, we must add Lock to every module which will be operated by threads, even the FileList Dialog. These Locks will make the program looks very bad.

3 Observer Pattern

We use the “observer pattern” to solve the problem listed above. We add two new modules : a new top-layer “SynDBHandler.py” to the database and a “DBObserver.py”.

In the new design, when a module want to read or write the data from database, it will first get a instance of SynTorrentDBHandler (SynDbHandler.py) and can register a “updatefunction” to TorrentObserver(DBObserver.py).

When a ”checking thread” get a new status of a torrent. it will get a new instance of SynTorrentDBHandler (SynDbHandler.py) and call the SynTorrentDBHandle.addTorrent() to write the new status into the database. In SynTorrentDBHandle.addTorrent(), it first Call the TorrentDBHandler.addTorrent() (CacheDBHandler.py) to update the status of torrent, then it will call the TorrentObserver.update() to update all the modules who have registered in observer.

For ”checking threads”, they only call TorrentDBHandler.addTorrent() to update the status of the

torrent. They don't know the existence of observer and modules which have to be updated. This will reduce the connections between modules and "checking threads".

For the synchronization problem, because it is TorrentObserver who actually call the update function of modules, So if we add a single Lock to TorrentObserver, we can solve the synchronization problem, and don't have to add Lock in the modules such as "abcfileframe.py"

Another advantage of this design is that the old modules (such as BuddyCast) who use the TorrentDBHandler.addTorrent() (CacheDBHandler.py) to update the status of torrent are still compatible with the new design. They can still work without any change, just don't have the synchronizing feature.