discussed was to use http/webdav for the transport, possibly reusing an existing http server implementation such as Apache, but this approach did not get much support.

A central notion in Akonadi is that it constitutes the one central cache of all PIM data on the system and the associated metadata. Whereas the old framework assumed online access to the storage backends to be the normal case, Akonadi embraces the local copy that has to be made as soon as the data needs to be displayed to the user, for example, and tries to retain as much of the information it has already retrieved as possible, in order to avoid unnecessary redownloads. Applications are expected to retrieve only those things into memory that are actually needed for display (in the case of an email application, the header information of those few email messages that are currently visible in the currently shown folder, for example) and to not keep any on disk caches of their own. This allows the caches to be shared, keeps them consistent, reduces the memory footprint of the applications, and allows data not immediately visible to the user to be lazy-loaded, among other optimizations. Since the cache is always present, albeit potentially incomplete, offline usage is possible, at least in many cases. This greatly increases the robustness against problems with unreliable, low-bandwidth, and high-latency links.

To allow concurrent access that does not block, the server is designed to keep an execution context (thread) for each connection, and all layers take a potentially large number of concurring contexts into account. This implies transactional semantics of the operations on the state, proper locking, the ability to detect interleavings of operations that would lead to inconsistent state, and much more. It also puts a key constraint on the choice of technology for managing the on-disk persistence of the contents of the system, namely that it supports heavily concurrent access for both reads and writes. Since state might be changed at any time by other ongoing sessions (connections), the notification mechanism that informs all connected endpoints of such changes needs to be reliable, complete, and fast. This is yet another reason to separate the low-latency, low-bandwidth, but high-relevance control information from the higher-bandwidth, higher-latency (due to potential server roundtrips), and less time-critical bulk data transfer in order to prevent notifications from being stuck behind a large email attachment being pushed to the application, for example. This becomes a lot more relevant if the application is concurrently able to process out-of-band notifications while doing data transfer at the same time. Although this might not be the case for the majority of applications that currently exist for potential users of Akonadi, it can be reasonably assumed that future applications will be a lot more concurrency-aware, not least because of the availability of tools such as ThreadWeaver, which is discussed in the next section of this chapter. The high-level convenience classes that are part of the KDE-specific Akonadi access library already make use of this facility.

Another fundamental aspect of Akonadi already present in this first iteration of the design is the fact that the components providing access to a certain kind of storage backend, such as a groupware server, run as separate processes. This has several benefits. The potentially error-prone, slow, or unreliable communication with the server cannot jeopardize the stability of