the overall system. Agents* can crash without taking down the whole server. If synchronous interaction with the server is more convenient or maybe even the only possible way to get data from the other end, it can block without blocking any other interaction with Akonadi. They can link against third-party libraries without imposing dependencies on the core system, and they can be separately licensed, which is important for those cases where access libraries do not exist as Free Software. They are also isolated from the address space of the Akonadi server proper and thus less of a potential security problem. They can more easily be written by third parties and deployed and tested easily against a running system, using whatever programming language seems suitable, as long as support for DBUS and IMAP is available. The downside, of course, is the impact of having to serialize the data on the way into the Akonadi store, which is crossing process boundaries. This is less of a concern in practice, though, for two reasons. First, it happens in the background, from the user's perspective, without interrupting anything at the UI level. Second, the data will either be already available locally, in the cache, when the user asks for it, or it will be coming from a network socket that can pass the data onto the Akonadi server socket unparsed, in many cases, and potentially even without copying it. The case where the user asks to see data is one of the few that need to be as fast as possible, to avoid noticeable waiting times. In most cases the interaction between agents and the store is not performance-critical.

From a concurrency point of view, Akonadi has two layers. At the multiprocessing level, each user of Akonadi, generally an application or agent, has a separate address space and resource acquisition context (files, network sockets, etc.). Each of these processes can open one or more connections to the server, and each is represented internally by a thread. The trade-off between the benefits of using threads versus processes is thus side-stepped by using both: processes where the robustness, resource, and security isolation is important, and threads where the shared address space is needed for performance reasons and where the code is controlled by the Akonadi server implementation itself (and thus assumed to be less likely to cause stability problems).

The ability to add support for new types of data to the system with reasonable effort was among one of the first desired properties identified. Ideally, the data management layer should be completely type agnostic, and knowledge about the contents and makeup of the data should be centralized in one place for each kind of data (emails, events, and contacts initially, but later notes, RSS feeds, IM conversations, bookmarks, and possibly many more). Although the desire to achieve this was clear from the start, how to get there was not. Several iterations of the design of the core system itself and the access library API were needed until the goal was reached. We describe the end result a bit further on in this section.

During the initial architecture discussions, the group approached the big picture in a pretty traditional way, separating concerns as layers from the bottom up. As the diagram of the white

* Entities interacting with the Akonadi server and reading and writing its data are referred to as *agents*. A general example would be a data mining agent. Agents that deal specifically with synchronizing data between the local cache and a remote server are called *resources*.