LHSG-CT-2003-503420

BioXHIT

A European integrated project to provide a highly effective technology platform for Structural Genomics.

Life Sciences, Genomics and Biotechnology for Health

WP5.2: De 5.2.3 Specification for Version 1 of the Project Database Handler design

Due date of deliverable: **31.12.2005**
Actual submission date: **31.12.2005**

Start date of project: **1.1.2004** **Duration: 60 months**

Organisation name of lead contractor for this deliverable: CCP4/CCLRC Daresbury **Author:** Peter Briggs

WP 5.2: Data management and project tracking in structure solution software. Coordinator Peter Briggs (Partner 10), contributing Partners 1C, 10.

De 5.2.3 *Specification for Version 1 of the Project Database Handler design*

Software pipelines such as those outlined in Section 4 will require a component to manage data flow and track project history. This is important in order to make information from all preceding steps available to software components in the pipeline. This WP is concerned with the implementation of a project-tracking database, which can be used within the context of a structure determination software pipeline in order to:

(i) keep track of progress through the structure solution pipeline

(ii) review progress when the pipeline is running in service mode. The owner of the project may wish to query progress through the pipeline, for example a biologist makes a request to the crystallisation service to see whether protein crystals have yet been obtained

(iii) diagnose problems or failures (particularly important for automated procedures). For example if a refinement fails, manual intervention may be required to trace back whether the appropriate model was used in molecular replacement, or whether there were problems in the data processing

(iv) improve procedures by identifying their strengths and weaknesses

REPORT on the Specification for Version 1 of the Project Data Handler Design

1. Introduction

This report outlines the specification for version 1 of the project database handler, which is deliverable D 5.2.1. The specification includes details of the three key components that compromise the full system.

This project aims to provide the following:

- A data store for storing and retrieving the information required by and generated from the process of determining macromolecular structures using the technique of X-ray crystallography, from any point after the user has a set of images obtained from a diffraction experiment. The data store will enable the progress of the structure determination to be tracked and reviewed.
- A brokering application or "handler" that will mediate interactions between the data store and any client applications that wish to store or retrieve information.
- Tools for displaying the information held in the data store ("visualisation tools" or "visualisers")

Not all the requirements have been specified yet. This report therefore outlines the current specification. Later versions of the system are likely to have revised specifications for one or more of these components. To this end, the original specification for the proto-handler is included as an appendix to this report.

2. General Concepts

2.1 Project Database

The project database will ultimately consist of three components:

- **Project History/Tracking Database:** this will store information about the progress of the project by recording the information associated with runs of

programs or other applications. The current CCP4i database is a simple tracking database.

- **Project Knowledge Base:** this will store information about the project as a whole, for example data that is known from steps prior to entering the software pipeline (the type of diffraction experiment performed or the sequence of the target protein) and data that is gathered as a result of running software (for example estimates of the solvent content). There is no project knowledge base equivalent currently implemented within CCP4i.
- **Operational Database:** this will store "operational" or "working" data - data objects that are used by software during the lifetime of some part of the structure solution process. These objects are characterised by being *volatile* (i.e. short lifespan) and by the fact that they may not be in universally recognised formats (e.g. pickled python objects used by a particular application to store working data).

The three databases are separate but related - as programs are run they may wish to store operational data, the resulting jobs are stored in the tracking database, and the output data may be stored in the project information database.

The three databases are illustrated schematically in figure 1 below:

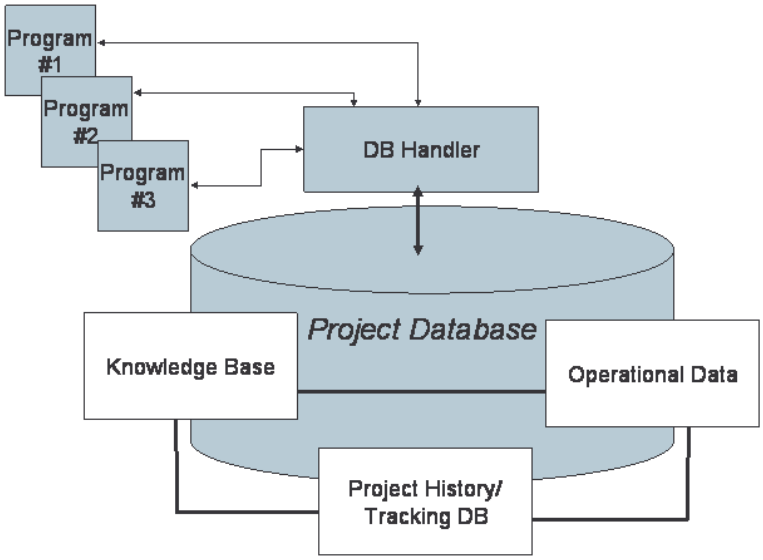


Figure 1: schematic representation of the database components

Issues that need to be resolved include what data needs to be stored, and how. For example: within CCP4i a project consists of a directory which contains a "database file" which stores details of each job run, in addition the directory also contains data files which have been imported or generated during the structure determination process. To begin with the simplest possible database structure will be implemented.

2.2 Project Database Handler

The project database handler will provide a mechanism for **applications** (programs or scripts) to store and recover data from a common place without having to know about the implementation details of the data storage (i.e. how the data is actually stored). It will act as a server and handle different client applications accessing a single database simultaneously, as illustrated schematically in figure 2 below:

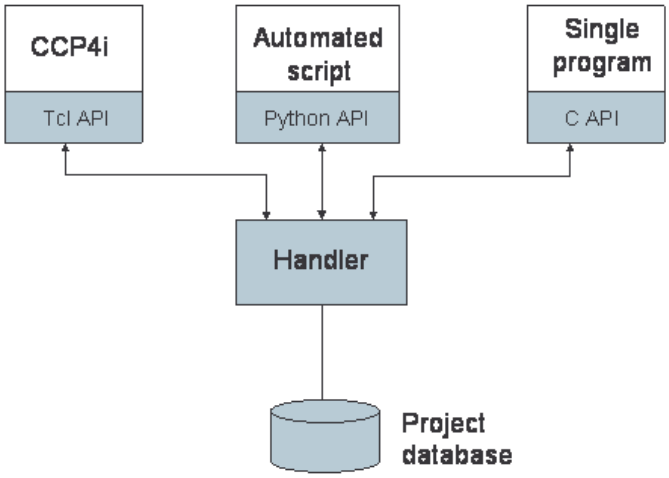


Figure 2: schematic of the project database handler

The components shown in figure 2 are:

- The **application** which can be a program, script or other executable module
- The **project data handler** which receives requests from the application to manipulate the data stored in the database(s)
- The **project database** which stores the data in some storage system

A more realistic illustration of the architecture of these three basic components of the system and their interactions are represented schematically in figure 3.

The following components will be provided:

- **Client API layer**: a library of functions for use by application programs to communicate with the handler (implemented in a number of different programming languages or otherwise made accessible to them). The client API is detailed in section 3.3.3.
- **Database**: a database implemented in some system that is actually used to store the data (e.g. the current CCP4i "flat file" format, or a mySQL database, or some other system). The initial database specification is detailed in section 3.3.
- **Handler**: a program which accepts commands (*requests*) in a standard format from external applications which tell it how to manipulate the data stored in the database, and issues *responses* to those requests

The handler itself consists of three further components:

- An *external communication layer* that takes requests received from the application (and issues responses).
- A layer which *verifies* the requests and *translates* them from the external format into a generic internal format i.e. the generic database API
- A *generic database API* layer which implements the functionality required to access the database in a variety of specific database languages - one for each type of database (mySQL, flat file etc) - which the handler supports.

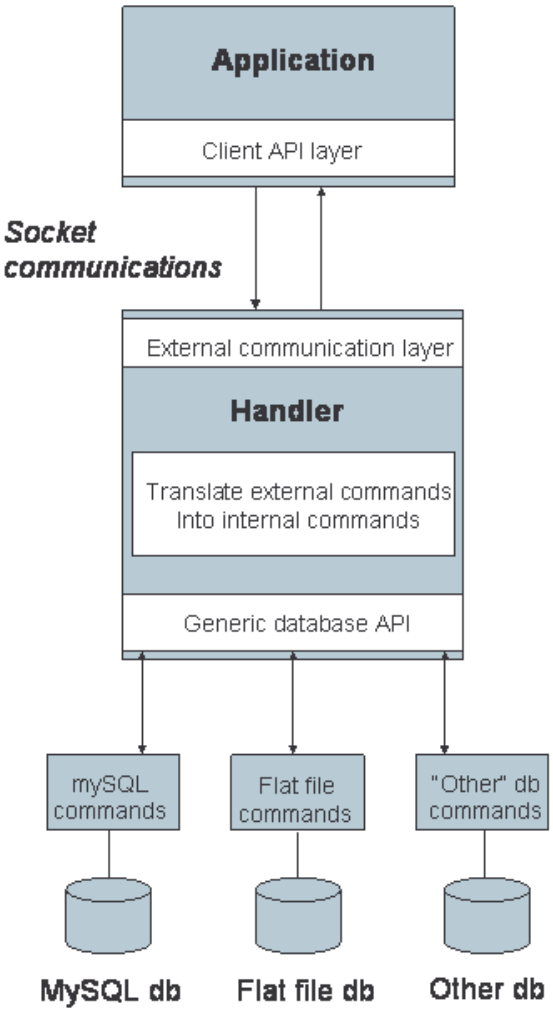


Figure 3: detailed architecture of the project database system

2.3 Visualisation Tools

Visualisation tools are applications that take the information stored in the data store and present different views that help the end user to better understand relationships within that data.
A simple example of a visualisation tool is the job list within CCP4i - this shows some basic attributes of each job within a project and allows the user to interact with the database by selecting jobs and then performing operations such as listing and viewing the associated input and output files, or rerunning the task with the same parameters.
More sophisticated tools can be envisaged, for example: showing the job list graphically as a "network diagram", which would make it easier to see pathways through the structure solution process and understand how the final result was arrived at.

3. Version 1 Implementation

3.1 Components

There are four components that need to be implemented:

1. Database handler
2. Tracking database schema
3. Client API library to communicate with the handler
4. Basic visualiser

3.2 Version 1 Handler

The handler will not consider security and authentication issues.

3.3 Version 1 Tracking Database

Initially we will use a minimal database as the data store. This can be thought of as having a hierarchical structure: the database will contain one or more *projects*, each of which will contain one or more *jobs*. Each job can have one or more associated *facts* and one or more associated *dingbats*.

These database elements are described more fully in table 1.

Table 1: Descriptions and attributes of the basic database elements

Database Element	Description	Attributes
Project	A collection of jobs that have something in common, for example steps towards determining the structure of a single protein. Project names must be unique.	• project id (unique) • creation date • project name (unique)
Job	A job is an instance of an operation, such as a run of a program or script.	• job id (unique) • creation date • parent project id
Fact	A fact is some item of information associated with a job, for example the solvent content or cell parameters. Note that facts do not have to have unique names.	• fact id (unique) • insertion date • name • value • parent job id
Dingbat	A dingbat is storage of some complex object, for example a pickled python object. Note that dingbats do not have to have unique names. Issues: The database will store dingbats as base64 encoded strings. The application API layer must transform the input object from the application into a suitable form before transmission to the handler for storage. The application API layer must also perform the reverse transformed when retrieving the encoded object from the database.	• dingbat id (unique) • insertion date • name • description • type • value • parent job id

The relationships between the database elements are represented schematically in the figure 4 below.

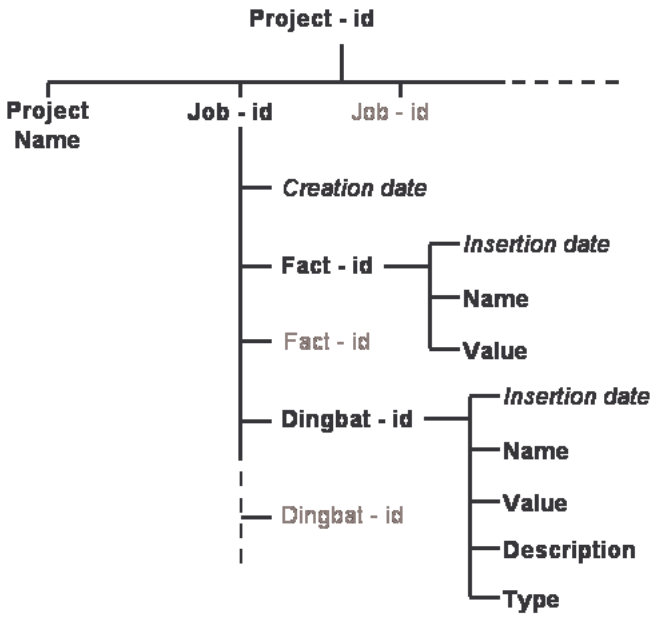


Figure 4: relationships between the database elements

Functions for the handler to interact with this "generic" minimal database are outlined below in section 3.3.1. The generic minimal database and the generic functions will initially be implemented in mySQL.

3.3.1 Generic Database API Functionality

The API to the database within the handler will provide the core functionality required to manipulate the data outlined above in section 3.3. The API functions are outlined in tables 2 and 3 below.

Table 2: Functions defining the basic DB API for the handler

Function	Description	Example Implementation
Open a project	Create a new project database (or open an existing database)	OpenProject *project*
Create a new job	Create a new job id within the specified project and set the creation date automatically to be the date and time that the handle was created. Return the job id.	*job_id* = NewJob *project*
Store a fact	Create a new fact for a specified job in a specified project. The fact will have *name* and *value* attributes specified by the calling application. The insertion date for the fact will automatically be set to the date and time that the new item was stored.	SetData *project job_id name value*

Retrieve a fact	Request the value associated with the most recent fact stored in a specific project for a specific job and specific name. "Most recent" means the fact with the latest insertion date. Return the matching value.	*value* = GetData *project job_id name*
Import a dingbat into the database	Create a new dingbat for a specified job in a specified project. The dingbat will have *name*, *description*, *type* and *value* attributes specified by the calling application. The insertion date for the dingbat will automatically be set to the date and time that the new item was stored.	ImportDingbat *project job_id name description type value*
Retrieve the description and type information of a dingbat	Fetch the description and type attributes of a dingbat for a specified job in a specified project.	*description_and_type* = DescribeDingbat *project job_id name*
Export a dingbat from the database	Fetch the value associated with the most recent dingbat stored in a specific project for a specific job and specific name. "Most recent" means the dingbat with the latest insertion date. Return the matching value.	*object* = ExportDingbat *project job_id name*
Delete a dingbat from the database	Remove a dingbat and its associated data from a specific project for a specific job and specific name.	DeleteDingbat *project job_id name*

Table 3: Functions defining the extended DB API for the handler

Function	Description	Example Implementation
List projects	Return a list of names of all the projects in the database.	*project_list* = ListProjects
List jobs	Return a list of job ids in a specified project	*job_list* = ListJobs *project*
List facts	Return a list of the facts associated with a specific job in a specific project	*fact_list* = ListFacts *project job_id*
List dingbats	Return a list of the dingbats associated with a specific job in a specific project	*dingbat_list* = ListDingbats *project job_id*

3.3.2 External Communication Layer Functionality

These are the commands recognised by the handler when they are received from an external application. The handler will offer a set of **database commands** that map directly onto the functions outlined in the Generic Database API Functionality section above. It will also offer a set of **administrative commands** that can be invoked by the application to perform the tasks outlined in table 4.

Table 4: Administrative functions available within the handler

Function	Description	Example
Connect to (log into) the handler	The application provides information that tells the handler e.g. which user it is operating on behalf of. This is like e.g. logging into a remote computer and having to supply username and password information.	Connect
Disconnect from (log out of) the handler	An application which is logged in can terminate the connection (this is like typing "exit")	Disconnect
Shut down the server	Tell the handler to terminate all client connections and then stop running.	Shutdown

The mode of operation that is envisaged is as follows:

- An application connects to the handler and logs in
- The handler listens for a request from the connected application
- When a request is received it verifies the request and acts upon it. The requests will correspond to one of the commands/functions outlined above:
    - A database request is translated from the external API format to the generic database API format and executed, causing some operation to be performed on the database, or
    - An admin request is executed in an admin layer
- The handler sends back the result of the operation (if appropriate) to the client application
- The handler listens for the next request and so on

3.3.3 Client API Library Functionality

The **client API library** is a set of functions that will be provided to applications to allow them to communicate with the handler.

Communications with the handler will be via a set of commands that the handler recognises. These commands will be packaged in some protocol that allows the handler to verify them (for example HTTP). The client API library will provide commands in different languages that will do the following:

- Open a client socket to the handler
- Issue a specific command recognised by the handler using the appropriate protocol
- Deal with any response from the handler and pass the results back to the application level
- Possibly also close the client socket

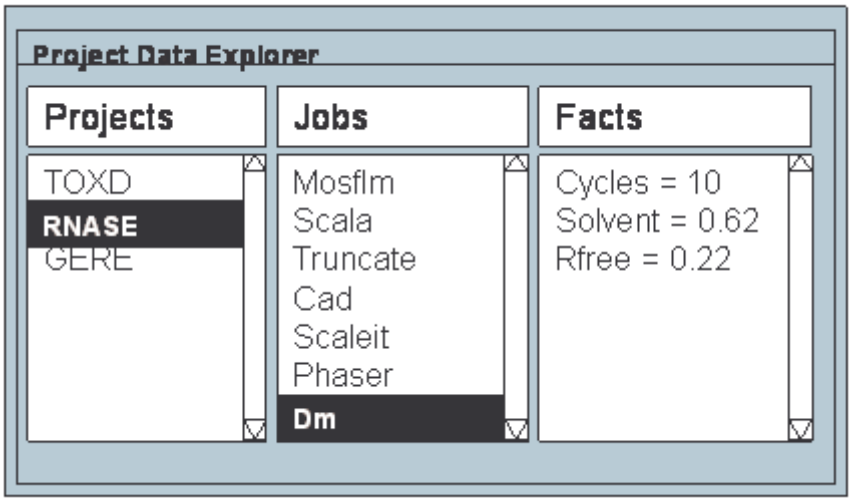
The client API will also perform any transformations of the data (e.g. base64 encoding/decoding) that are required, and will offer a range of functions that directly map onto the functions offered by the handler.

3.4 Prototype Visualiser: "Project Data Explorer"

The prototype visualiser is a client application that will provide a graphical interface to allow a user to explore the project information held in the database. The explorer

would provide three windows, one each for listing projects, jobs and facts. The user can select a project name and the list of jobs is updated to show those jobs within that project. The user can then select a job and the list of facts is updated to show the facts for that job.

A cartoon of what the explorer might look is shown below:



The explorer will need to be synchronised with the database. This could be done in one of two ways:

- The explorer performs a "refresh" operation either prompted by the user (e.g. hitting an update button), or at regular time intervals; or,
- The explorer stays connected to the handler for its lifetime and listens for broadcasts from the handler that prompts it to update (see section 4.2 for information about broadcast communications).

4. General Issues

4.1 Modes of Use

Currently only one mode of use is envisaged, with a single user running a single handler on a single machine. The user may run one or more applications that can communicate simultaneously with the handler to store and retrieve data from the database.

Two broad modes of use are envisaged:

- **Manual structure determination**: for example users running programs via CCP4i. In this case each job is initiated by a user who may also made requests to browse data in or add to the database
- **Automated systems**: where there is minimal user intervention with database requests, and decisions are made automatically by a script or other pipeline.

The open architecture specified in this report should however be readily extensible to more sophisticated modes of operation, for example multiple users accessing the same project database.

4.2 Communications Protocols

Communications between the handler and the applications take a number of forms:

- **Requests** sent from an application to the handler
- **Responses** to those requests sent from the handler to the application
- **Broadcasts** sent by the handler to the application which is not a response to a request

The CCP4i prototype handler needed broadcast communications, as a way for the server to inform the application of changes e.g. updates to the database. This may be required for client applications that maintain a persistent connection with the handler, but is not considered an essential part of the version 1 specification.

The communications between the client applications and the handler (via the client API) need to be defined. There are two components:

- Command/response language: the syntax and vocabulary of the requests, responses and broadcasts sent via the sockets between the client application and the handler
- Communication protocol: how these commands are "wrapped up" in order to be sent across the sockets.

Wrapping is important as it provides a mechanism for validating the syntax of communications, and parsing the different components (for example, separating the return value from the result of a request, and for distinguishing between responses and broadcasts). The version 1 specification will use a simple XML protocol (described in appendix A).

Other protocols for future consideration include:

- HTTP (hypertext transfer protocol)
- SOAP (Simple Object Access Protocol)

Appendix A: Specification for Version 1 of DB Handler Communication Protocols using XML

A.1. Overview

There are two components: the command language (the messages that are being sent/received) and the communication protocol (the wrapping). Section A.2 treats the former and section A.3 the latter.

A.2. Command language

ClientAPI.py (CVS version 1.9) defines a set of commands that map onto numbers which are passed to the handler. The following mappings are in place:

0: OpenDB *host username password dbname*
1: NewProject *projectname*
2: NewJob *projectid jobname*
3: SetData *jobid name value*
4: GetData *jobid name*
5: ListProject
6: ListJob *projectid*
7: ListItem *jobid*
8: ImportDingbat *jobid name description type value*
9: ExportDingbat *jobid name*
10: DeleteDingbat *jobid name*
11: DescribeDingbat
12: GetProject
13: Version

I would suggest that the most immediate thing to do would be to substitute the numbers (i.e. 0, 1, 2...) with the names of the equivalent commands in the client API (i.e. OpenDB, NewProject, NewJob...). This would need only minor changes to ClientAPI.py and server.py.

A.3. Protocol

We previously identified three types of messages being passed between the client and the server:

1. requests (from the client to the server, to perform some action)
2. responses (from the server to the client, to report the result of a request)
3. broadcasts (from the server to the client, to report some change in status of the database and not solicited by the client)

Only “requests” and “responses” need to be implemented in the initial version. Requests need to specify a command and a set of arguments. Responses need to return a value and a status.

The first attempts passed raw strings through sockets. This should be extended to using some kind of simple (pseudo) XML. There are at least two possible ways of doing this:

One possibility is to wrap the command and the arguments with specific tags that identify the individual parts, for example:

```
<db_request command="OpenDB">
```

```
        <argument name="host">hostname</argument>
        <argument name="user">username</argument>
        ...
    </db_request>
```

and:

```
    <db_response status="OK">value</dbresponse>
```

An alternative approach is not to use attributes but tags only, e.g.:

```
    <db_request>
        <request_name>OpenDB
        <host>hostname</host>
        <user>username</user>
        ...
        </request_name>
    </open_db_request>
```

and:

```
    <db_response>
        <status>OK</status>
        <result>value</value>
    </db_response>
```

The precise form of the command mark-up should be documented in the supporting documentation for the implementation of the version 1 handler, and be resolved in a future version of the specification.

A.4. Implementing the protocol

This will require two new functions:

- to encode (= wrap a "plain" command or response in XML)
- to decode (= recover it from the XML).

ClientAPI functions will encode the arguments from the user and send the encoded string to the server; the server will decode the string and attempt to execute the command. It would then encode the response and send that back to the ClientAPI, which would then decode it and give the result back to the client application.

Applications will do not use ClientAPI to communicate with the handler must deal with the encoding/decoding themselves.

Appendix B: Unresolved Issues List for the Database Handler

This appendix summarises the various issues that the version 1 handler is intended to resolve.

Name	Description	Resolution plan
CCP4i backend	The CCP4i flat-file database backend is fundamentally different from the mySQL implementation, and so the implementation needs to be documented. The flat-file backend will most likely need to be reimplemented (and possibly not in Tcl).	
Client attributes	What attributes do client applications have? For example, do clients operate on behalf of users? Do clients connect synchronously or asychronously to the handler? Do clients need to "watch" the database? Are there different types of client?	Identify different types of client and client attributes via "use cases" or similiar
Communication protocols	How are requests and responses passed between the application and the handler? This includes dealing with the names of the requests that are passed between the handler and the clients.	
Connection persistence	How persistent does the connection between the client and the handler need to be?	
Database backending	Does the handler application need to deal with arbitrary abstract database backends, for example ISpyB? That is, does the handler need to be able to retrieve data from an arbitrary database with a different database structure, items etc than that implemented by this project?	
Data persistence	How persistent is the data stored in the database? Some data may only be stored for the lifetime of a job, other data may be required afterwards.	Identify how data will be used via "use cases" or similar
Data transfer	How is information transferred between different databases (e.g. facility databases, LIMS for wet labs, the tracking database). How do these overlap or complement each other in terms of scope etc?	Begin dialogue with other interested parties about their requirements
Data types	What kind of data are we storing (e.g. items of text? compound data items such as cell parameters? more sophisticated objects such as python pickles?)	
Error Checking	What error checking needs to be performed? (e.g. checking that project names are unique) Where is this checking performed? (e.g. in the client side API, in the handler)	
Installation	How difficult is it for non-expert user to install and use the system?	

Languages	What languages do we need to provide APIs for? (Python, Tcl, C? Java?). Are there any language-dependent features in our implementation?	
Performance	What volume of data can the system handle (store, transfer)? Are there issues with speed of access, response times etc?	
Portability and dependencies	What platforms can the handler and database run on? What packages does the system depend on to run? (e.g. python and mySQL) What are the minimum versions required from each dependency?	
Security	Security issues include: permissions on data for individual users and/or client applications; protecting against malicious attacks; authentication (checking that the requests it receives are from a trusted source)	
Testing	How do we test the handler? What are we trying to test?	Identify areas to test by considering the requirements for the handler and the database?
Tracking	How do we incorporate the concept of subjobs? i.e. a job which is the child of a previous job?	

Appendix C: Original Database Handler Specifications

This document outlines the proposed specification for a CCP4i database handler that was started in 2003 but never fully completed. However the development of that prototype has informed many of the specifications in this report.

Aims: Provide an API to the CCP4i project history database that can be accessed by: the main CCP4i process; independent processes started by the main CCP4i process (``jobs''); external (non-CCP4i) applications.
In the first instance this must reproduce the existing functionality within CCP4i v1.3.8 for interacting with the database. It must be able to interact with the existing format of the database (currently a flat format CCP4i-parameter or "def" file). However it should also be extensible, that is, be able to accommodate a wider scope of commands and to be easily extended to use different database backends.

Core Requirements

1. Reproduce existing functionality found in CCP4i 1.3.8.
2. Clearly separate general database functionality from CCP4i-specific functionality.
3. Provide an API which can be used by non-CCP4i applications to interact with the database (applications may be in languages other than Tcl).
4. Provide an API which can be used in a distributed computing environment.

Additional Requirements

5. Provide a way of expanding the scope of the project history database (allow storage and access of new data items).
6. Enable new/different database backends to be used for storing the project history database.
7. Enable updates/changes to item definitions.
8. Enable access to external/3rd party databases (for example LIMS).

General Issues

Security and authentication are issues that will need to be addressed if the server is accessible over a network. These issues have not yet been explored.

Component-specific Issues

1. **Requirements for CCP4i**

   The main CCP4i process needs both read and write access to the database, as it needs to be able to register new jobs, set/edit associated information, and delete jobs from the database, as well as requesting information on the current state of the database. It must be able to handle ``update'' messages from the database handler when the status of the database changes.

2. **Requirements for Running Scripts (``jobs'')**

   Running jobs need to be registered with the handler as part of their startup, but after this point only require limited write access (to add output files, register script termination) and no read access to the database.

3. **Requirements for Non-CCP4i Applications**

Applications may be written in any language, so the protocol for exchanging requests/information between the handler and the applications needs to be as generic as possible.

4. **Requirements for accessing other Databases**

   These are not currently known.

Current Implementation

The database for a given project is stored on disk as a flat file in CCP4i parameter file format (".def"). The data in the file is read into an array in the main CCP4i process when the project is first opened by the user. Subsequent queries or actions on the database are made via the array held in memory, which is periodically written out back to the file.
Functions for interacting with the database information are tightly embedded in the code for the main CCP4i process. In some cases general database functions are mixed together with CCP4i-specific code, thus the full range of functionality is not easily accessible by non-CCP4i applications.

Possible Solutions

The current implementation does not meet all the requirements, for example it doesn't provide an API which can be used easily by other applications. The current prototype solution uses a separate database handler process. Application programs interact with the database via socket requests made to the database handler process. This addresses the issues of networked operation, and removes the requirement for multiple language-specific APIs.

Other possible solutions (currently not under consideration):
It would be possible to separate the database handling API cleanly from the main CCP4i process in such a way that other applications could load the commands into an interpreter and use them to interact with the database directly. It would not be straightforward for non-Tcl applications to interact with the database - this could be addressed by implementing the API in a number of languages, however this would incur an undesirable maintenance overhead. Also it fails to address the issue of operating over a network.

Outline of the Database Handler (``DbCCP4i'')

The provisional name for the database handler process is DbCCP4i.
The current model of DbCCP4i consists of three basic sets of components:

1. Core handler. This manages the sockets for communicating with the connecting processes, performs basic bookkeeping (e.g. tracking which processes need access to which databases, which databases are currently open, and so on), and passes requests and responses between processes and databases via the appropriate API layers.
2. API layer to deal with requests received from and responses sent to the connected processes using the appropriate protocols (e.g. XML/http).
3. API layers to deal with different types of database, e.g.:
   i. CCP4i project database
   ii. SQL database
   iii. XML database

A DbCCP4i process can be started either by the main CCP4i process, or separately e.g. via a user command. Each CCP4 project history database should only be accessed by a single DbCCP4i at any one time (this could be controlled via lock

files), though a single DbCCP4i could access more than one project database - for example, a user is browsing the project history data in one project, but also has running jobs which are registered in a second project. To conserve system resources each user should only have one DbCCP4i running at anytime.

Ultimately it should be possible to allow several users to access the same database simultaneously via a single DbCCP4i.

Processes that wish to access the database must first register themselves with the DbCCP4i - in a secure environment this should include some authentication procedure. Registered processes may have different interaction requirements, for example: the main CCP4i process needs to able to read and write to the database, but it also needs to know when the database content changes (so it can update its display); whereas running jobs only need to send information to the database (new output files, job finishes or fails) and do not need to be informed of updates. There will need to be an "update" mode whereby the DbCCP4i broadcasts notice of an updated database to all connected processes which have registered an interested in knowing when certain types of database content have changed.

There needs to be a mechanism for new processes to detect and connect to an existing DbCCP4i when trying to access a project database - possibly through information stored in the lock file.

DbCCP4i processes should have a number of different persistence modes. Initially the DbCCP4i will persist as long as it has still has registered processes (processes should unregister themselves on shutdown). It should also be possible to leave the DbCCP4i running indefinitely and have processes connect and disconnect as required. There also needs to be a way to cleanly shutdown a DbCCP4i e.g. via a user command.

Prototype dbCCP4i: current status

A prototype version of dbCCP4i now exists which has been built on top of CCP4i 1.3.9.

Currently Unresolved Issues for DbCCP4i

1. Information exchange protocols: i.e. how are requests and information to be passed between the server and the application (syntax etc)?
2. Security: how to make sure that only authorised applications are allowed to interact with the database?
3. Locking issues: should multiple processes be allowed access (read-write, read-only) to the same database (queuing/lock-grab/lock-out)?

Work Breakdown for Implementation

1. Define the APIs for interaction between the DbCCP4i and the client processes.
2. Define the APIs for interaction between the DbCCP4i and the databases.
3. Rewrite CCP4i code so that CCP4i interacts with the database through a set of API commands. More specifically this consists of separating the existing commands into two distinct classes: a set which interact directly with the global `database' variable and its image on disk (the core database commands); and a set which interact with the database information via calls to the first.

4. Write the core DbCCP4i process and implement functions to allow processes to start-up, connect to/register with, disconnect/unregister from and shutdown the DbCCP4i. Investigate/deal with security issues.
5. Implement the project database handling functions identified in 3) inside the DbCCP4.
6. Rewrite the CCP4i database API to interact with the project history database via the DbCCP4i.
7. Document the implementation and usage of the DbCCP4i and APIs.
8. Perform local testing.
9. Distribute new CCP4i for wider testing.