ProServer installation guide

This short guide will describe how to install the ProServer Perl DAS server on a Unix-type system. It will also give the basics on how to write a source adaptor (or “plug-in”) for ProServer and how to modify the the ProServer code itself to provide some more customized fields in the response to some DAS request.

Most shell examples assumes that you are using a sh-type shell, such as `sh`, `ksh`, or `bash`.

Please note that the latest version of ProServer (version 2) look and function slightly different from what is described here due to recent updates made by the ProServer author, Roger Pettett. It is my intention to update this document to be in sync with the latest version of the software and to note any incompatibilities introduced between releases. For now though, the version described in this document is version 1 of the server.

1. Getting and unpacking the ProServer distribution

- Get the ProServer distribution tar archive from

  ```
  http://www.sanger.ac.uk/proserver/ProServer-1.tar.gz
  ```

  It is also available as part of the `Bio-Das2` CVS module on the `cvs.biodas.org` CVS server, but so is a lot of other things, and it is not easy to untangle the server from there unless you know what you’re doing, and that the HEAD revision of the code might differ considerably from what is described in this document.

  See

  ```
  http://cvs.biodas.org/
  ```

  for information on how to check out the `Bio-Das2` module, if you’re interested.

- Unpack the tar archive with

  ```
  $ tar xvzf ProServer-1.tar.gz
  ```

  If you are using a tar program that does not understand the `z` flag (like the native one on Tru64 for example), instead use

  ```
  $ gzip -d -c ProServer-1.tar.gz | tar xvf -
  ```

  This will create a directory called `ProServer` with a number of subdirectories.

2. ProServer distribution layout

These are the most important parts of the distribution:

```
ProServer-1/
|
+---eg/
|
‘---ProServer/
    |
    ‘---SourceAdaptor/
        |
        ‘---Transport/
```

- The `eg` directory contains the actual ProServer executable, and the configuration file.
- The `ProServer` directory (somewhat confusingly the same name as the top level directory of the tar archive) contains the Perl modules used by the DAS server.
- The `SourceAdaptor` subdirectory contains the source adaptor modules, and the `Transport` directory below it contains the transport modules that are used by some of the source adaptor modules. More on those later.
- The `Bio` and `Das` symbolic links that you might notice is just a workaround to avoid a deep directory hierarchy since the namespace for the ProServer Perl modules actually is `Bio::Das::ProServer::`. Ignore them.

3. Before starting ProServer for the first time

Before starting the server you will most likely have to modify a few things in the main executable ProServer script in the `eg` directory.

- If the script is not executable, you should make it so:

```
$ chmod +x proserver
```

- Modify the path after the `#!` on the first line of the script so that the correct Perl interpreter is used. On my private machine where Perl is installed as `/usr/bin/perl`, for example, I need to modify the first line of the script into

```
#!/usr/bin/perl -Tw
```

- Modify the `use lib` line a bit further down so that it reads

```
use lib qw( .. );
```

  (That's two dots in parenthesis.) Alternatively, provide the absolute path to the `ProServer` subdirectory of the server distribution.

  If you have installed Perl modules in non-standard locations, you might also want to add these to the list of directories to search. For example:

```
use lib qw( ..
  /home/user/ensembl-cvs/ensembl/modules
  /home/user/ensembl-cvs/ensembl-compara/modules );
```

One way to figure out what extra Perl modules need installing is to start the server. Do this by going to the `eg` directory and say

```
$ ./proserver
```

If the server dies with a message starting with

```
Can't locate some/module.pm [...]
```

then see to that the missing module is installed and that it is found by the Perl interpreter.

Some module that are not commonly found in a default Perl installation are the `Config::IniFiles`, `HTTP::Daemon`, and `Compress::Zlib` modules. It is common to want to serve data from MySQL databases with ProServer. To do this you also need to find the `DBI` and `DBD::mysql` modules. All these modules are available from CPAN, but if your OS distribution provides them as native packages, then it is highly recommended to install them from there instead.

4. Starting the server for the first time (with no data)

Once all the needed modules are installed, but before you try to modify any adaptors or configuration files, start the server to see what happens.

Move to the `eg` directory and type

```
$ ./proserver
```

If everything is ok, this will start the server in the background and return you to the command prompt.

ProServer is a self-contained server, meaning that it doesn't rely on Apache, Tomcat or any other type of service to handle connections.

The server listens by default to port 9000, but this can be changed to any other free port. It also starts up five child processes that takes turns to answer requests arriving on the port. This is so that the server can answer multiple requests in parallel. Each child, however, only handles ten request, it is then killed and respawned by the parent process. This way ProServer avoids potential memory leaks and keeps it from potentially blocking resources from other processes. The number of child processes, and the number of request that each child process should handle, is configurable as well.

You will find the process ID (PID) of the server in a file named

```
proserver.$(hostname).pid
```

This means that you can kill the server with the command

```
$ kill $(<proserver.$(hostname).pid)
```

The log messages produced by the server are written to the file

```
proserver.$(hostname).log
```

The following is the full log file after starting and stopping ProServer on a machine called `shibito.local`. No changes were made to the original configuration file.

```
Proserver v1.5 startup...
Listening on host:port shibito.local:9000
Started 5 child servers
Wrote parent PID to ./proserver.shibito.local.pid [PID: 21738]
Please contact this server at this URL: http://shibito.local:9000/
Killing children...
Received SIGTERM. Clean child 7073 exit
Received SIGTERM. Clean child 26664 exit
Received SIGTERM. Clean child 29617 exit
Received SIGTERM. Clean child 2918 exit
Received SIGTERM. Clean child 12720 exit
```

5. Server and adaptor configuration (ini-files)

ProServer uses conventional "ini-files" for holding its configuration. By default, the configuration file named `proserver.ini` will be read, but this can be changed by using the `-c` flag when starting the server:

```
$ ./proserver -c myconfig.ini
```

Like all ini-files, the configuration file is divided into named sections. Configuration options within each section consists of key–value pairs. The section named `[general]` is used for server options, while all other sections are used for the individual data sources that are served from the server.

The most common options used in the `[general]` section are

`prefork`
: The number of child processes that should be maintained for serving parallel requests. If the server is serving a large number of data sources, or to a large number of users, this value may need to be increased from the default of five.

`maxclients`
: The number of requests that each child process should handle before being killed and respawned. The default value for this setting is ten, and can probably be raised.

`port`
: The port number that the server will listen to. The default port is port 9000, but any free port above 1024 will do.

`logfile` and `pidfile`
: These gives you the option to explicitly give a name for the log file and/or PID file. If you are running multiple servers from one and the same directory on one machine, then you will want to use these. Otherwise you will get file name conflict between the running servers.

Each data source has its own configuration section, with the section name defining the data source name. The two configuration options that *needs* to be defined for each data source are

`state`
: This is either `on` or `off` depending on whether the source will be active or turned off.

`adaptor`
: This should be the name of the adaptor used by the data source (see later). The name will be prepended with the string `Bio::Das::ProServer::SourceAdaptor::` to form the name of a Perl package.

Additional configuration options may be added to the configuration section of a data source if needed by either the source adaptor module or by the transport module. For example, the `dbi` transport module (handles connections to Perl DBI databases such as MySQL) may be used by setting the `transport` configuration option to `dbi` and then also specifying values for the `host`, `username`, `dbname`, `driver`, and optionally the `port` and `password` options.

The ProServer distribution comes with a predefined configuration file that may be used as an example on how to configure a number of the available source adaptors and transports.

The values for a configuration option is available to the source adaptor in a straight-forward manner. To get the value for the option `myoption`, the code within the source adaptor module only needs to say

```
my $optvalue = $self->config()->{'myoption'};
```

where `$self` is a reference to the current source adaptor object.

The following is the (slightly modified) full configuration for the `textmining` data source served by the DAS server located at

```
http://www.ebi.ac.uk/das-srv/genedas/das
```

which, if given a HUGO ID of a gene, serves text fragments from PubMed abstracts which has been text-mined by a group at the EBI:

```
[textmining]
state           = on
adaptor         = textmining
transport       = dbi
; transport specific setting:
driver          = mysql
host            = mysql-ensembl1.ebi.ac.uk
port            = 3034
username        = someuser
password        = somepassword
dbname          = textmining
; adaptor specific settings:
maxfragments    = 100
```

6. About source adaptors and transports (the way things work)

One single DAS server may serve multiple data sources. With ProServer, a data source is always configured to use one *source adaptor*. A source adaptor is a Perl module that knows how to handle the data source-specific parts of a DAS request, such as fetching the features on, or sequence of, a segment (chromosome, clone or similar) from the underlying database given the name of the segment and possibly the range on the segment.

The Perl module which implements a source adaptor should always inherit from the ProServer module `Bio::Das::ProServer::SourceAdaptor` (hereafter referred to as just `SourceAdaptor`) and must lie in the `Bio::Das::ProServer::SourceAdaptor::` namespace, i.e. an adaptor called `dbadaptor` should be placed in the `Bio::Das::ProServer::SourceAdaptor::dbadaptor` package.

The `SourceAdaptor` module contains stubs for a number of methods (subroutines) that will be called depending on what DAS request the DAS client makes. It is up to the source adaptor to override the methods that corresponds to the DAS requests that should be supported by the adaptor.

6.1 Source adaptor methods and DAS requests

The methods in the `SourceAdaptor` module that may be overridden by the adaptor and the DAS requests that they correspond to are the following (we only describe the method corresponding to the `features` request in full detail as it is by far the most commonly used one):

`build_features`

Corresponds to the `features` DAS request.

The method recieves a reference to a hash containing the keys `segment`, `start`, and `end`. The `segment` key will, as its value, contain the name of the segment for which the request was made. Depending on the client that the DAS server was attached to, this might be the name of a chromosome, a contig, a scaffold, or some other type of large assembly structure. It might also be the name or ID of a protein or gene if the server was attached to a ProteinDAS or GeneDAS client. We will not discuss the possible problems with various coordinate systmes and assemblies here.

The `start` and `end` keys might not exist, but if they do then any feature overlapping the region that they define on the named segment should be returned. If they are not provided, then features present anywhere on the named segment should be returned.

The return value from the `build_features` method should be a list (not a list reference) of features. Each feature is a reference to a hash with a set of special key–value pairs closely corresponding to the XML tags of the response to the `features` DAS request (see the DAS specification).

The names of the most commonly used keys are

`id` The unique ID of the feature.

`type` The type of annotation (required).

`method` The method of annotation used (required).

`start` The start position of the feature on the named segment (required).

`end` The end position of the feature on the named segment (required).

`ori` The orientation of the feature (`+`, `-`, or `0`) (required by the DAS specification, but ProServer will default to `0`).

`note` A note (optional).

`link` A URL to a web resource containing further information, ideally about this feature in particular (optional).

Have a close look at the `das_features` method of the `SourceAdaptor` module for further possibilities, default values, and for how the various values are used to build the XML response to a DAS `features` request.

`sequence`

The method gets called for the DAS `dna` and `sequence` requests.

The method recieves a named segement and an optional range on that segment in the same way as the `build_features` method does, and it is assumed to return a reference to a hash containing the two keys `moltype` and `seq`. The sequence is stored as the value of the `seq` key, and the molecular type is stored as the value for the `moltype` key.

Note that it is only reference DAS servers that needs to provide the ability to serve sequence.

`build_types`

This method corresponds to the `types` DAS request.

As for most of the other methods, the `build_types` method recieves a segment and a range, but for this DAS request, these are optional. When no segment is supplied, the DAS server is required to return all available types.

The reply should be a list of references to hashes. Each hash consist of the keys `type`, `method`, and `category`. These have the same meanings as specified in the DAS specification (the `type` key corresponds to the `id` attribute of the `TYPE` tag). There is also a `count` key which, as its value, should contain the number of features of this type on the segment in question.

`build_entry_points`

This method takes care of the `entry_points` DAS request.

The method should return a list of entry points. Each entry point is a reference to a hash containing the keys `segment` and `length`. The `segment` key should, as its value, have the name of a valid segment, and the length of the segment should be stored as the value for the `length` key.

`stylesheet`

This method should return a string containing the XML for a DAS stylesheet when requested by the `stylesheet` DAS request.

No interpretation is made of the returned XML data by the rest of the ProServer code.

Have a closer look at the `SourceAdaptor` module for more on how these methods are called and how the XML is being created from the data returned from them.

6.2 Initializing a source adaptor

In addition to overriding one or more methods from the `SourceAdaptor` module, all source adaptors must implement a method named `init` which is called once before the adaptor is handed any DAS requests and which modifies the `capabilities` attribute of the source adaptor object so that it reflects the capabilities provided by the adaptor. The capabilities are simple key–value pairs with the capability name as the key and the capability version (as defined by the DAS specification) as the value. See the example code further down for an example of this.

The `init` method may also do any additional tasks that might be required to set up the adaptor. If additional data needs to be stored in the source adaptor object, such as possibly large objects that only needs to be allocated once during the life-time of the adaptor object, this data should be stored in a separate private attribute. The name `_private` does not collide with any already existing attributes in the source adaptor object. The following is an example of this where a fairly standard cache module is used to provide an adaptor with the possibility to use a file cache for caching results:

```
$self->{'_private'}{'cache'} = Cache::FileCache->new(
    {
        'namespace'           => 'myadaptor',
        'default_expires_in'  => 12 * 3600,     # 12 hours
        'auto_purge_interval' => 600,           # 10 minutes
        'auto_purge_on_set'   => 1,
    }
);
```

6.3 Transports

A ProServer source adaptor is free to include the Perl `DBI` module to set up a connection to a database, and then to query that database from its `build_features` method by preparing and executing SQL queries, but it is also possible to use an already existing transport module instead to achieve the same effect.

A transport module, such as the `dbi` transport for MySQL etc. or the `getz` transport for SRS, hides the setting up and querying of the underlying database.

We have already shown an example of how to configure a source adaptor to use a transport. In the source adaptor code, the transport is then available and may be use in the following manner:

```
my $transport = $self->transport();               # get dbi transport
my $statement = qq( SQL QUERY TEXT );             # specify a query
my $rows      = $transport->query($statement);    # perform query
```

This example assumes that we're using the `dbi` transport and that `SQL QUERY TEXT` is the statement that we want to execute against the database to get the data needed to build the requested features. The returned value from the `query` method of the `dbi` transport is the same as from the Perl `DBI` method call

```
$sth->fetchall_arrayref({ });
```

where `$sth` is a Perl `DBI` statement handler. This means that for this particular source adaptor, it is a matter of looping over the rows to build the array of features:

```
my @features = ();

foreach my $row ( @{$rows} ) {
    my $feature = {
        # Pull fields out from the $row hash
        # reference into the appropriate keys of
        # the $feature hash reference.
    };
    push @features, $feature;
}

# ...

return @features;
```

7. Writing a source adaptor (simple example)

The following code is the full implementation of `mysimpledb`, a variation of the `simpledb` source adaptor module that is available in the ProServer distribution. It assumes that it has access to a MySQL database containing a table with the columns `segmentid`, `featureid`, `start`, `end`, `type`, `method`, `note`, and `link`. The name of the database table is configurable (see later).

The adaptor implements the `features` DAS request by overriding the `build_features` method, and the `types` request by overriding the `build_types` method from `SourceAdaptor`. Any feature overlapping the requested region on the segment will be returned (the `simpledb` adaptor in the ProServer distribution only returns features contained within the region). If no range is requested, all features on the segment are returned.

```
package Bio::Das::ProServer::SourceAdaptor::mysimpledb;

use strict;
use warnings;

use base qw(Bio::Das::ProServer::SourceAdaptor);

sub init
{
    my ($self) = @_;
    $self->{'capabilities'}{'features'} = '1.0';
    $self->{'capabilities'}{'types'}    = '1.0';
}

sub build_features
{
    my ( $self, $args ) = @_;

    my $segment       = $args->{'segment'} || return ();
    my $segment_start = $args->{'start'};
    my $segment_end   = $args->{'end'};

    my $dbtable = $self->config()->{'dbtable'} || return ();

    my $bounds_part = '';
    if ( defined($segment_start) && defined($segment_end) ) {
        $bounds_part = qq(
            AND     ((start >= '$segment_start'
            AND       start <= '$segment_end')
            OR       (end   >= '$segment_start'
            AND       end   <= '$segment_end'))
        );
    }

    my $query = qq(
        SELECT  featureid, start, end, type, method, note, link
        FROM    $dbtable
        WHERE   segmentid = '$segment'
        $bounds_part
    );

    # (continued on the next page)
```

```
    # (continued from the previous page)

    my $rows     = $self->transport()->query($query);
    my @features = ();

    foreach my $row ( @{$rows} ) {
        push @features,
          {
            'id'     => $row->{'featureid'},
            'start'  => $row->{'start'},
            'end'    => $row->{'end'},
            'type'   => $row->{'type'},
            'method' => $row->{'method'},
            'note'   => $row->{'note'},
            'link'   => $row->{'link'}
          };
    }

    return @features;
}

sub build_types
{
    my ( $self, $args ) = @_;

    my $segment       = $args->{'segment'};
    my $segment_start = $args->{'start'};
    my $segment_end   = $args->{'end'};

    my $dbtable = $self->config()->{'dbtable'} || return ();

    my $bounds_part  = '';
    my $segment_part = '';
    if ( defined $segment ) {
        $segment_part = "WHERE segmentid = '$segment'";

        if ( defined($segment_start) && defined($segment_end) ) {
            $bounds_part = qq(
            AND     ((start >= '$segment_start'
            AND       start <= '$segment_end')
            OR       (end   >= '$segment_start'
            AND       end   <= '$segment_end'))
            );
        }
    }

    my $query = qq(
        SELECT  COUNT(*) AS 'count', type, method
        FROM    $dbtable
        $segment_part
        $bounds_part
        GROUP BY type, method
    );

    # (continued on the next page)
```

```
    # (continued from the previous page)

    my $rows  = $self->transport()->query($query);
    my @types = ();

    foreach my $row ( @{$rows} ) {
        push @types,
          {
            'type'   => $row->{'type'},
            'method' => $row->{'method'},
            'count'  => $row->{'count'}
          };
    }

    return @types;
}

1;
```

This code goes into a file called `mysimpledb.pm` which is then put within the `SourceAdaptor` subdirectory of the ProServer distribution, or into some other place where the Perl interpreter might find it.

The configuration of a server that wants to include a source using this adaptor might look like this:

```
[mysource]
state           = on
adaptor         = mysimpledb
transport       = dbi
; transport specific setting:
driver          = mysql
host            = thedbhost.example.com
port            = 3034
username        = someuser
password        = somepassword
dbname          = dasdb
; adaptor specific settings:
dbtable         = das_features7
```

8. Modifying the VERSION of a SEGMENT

Within the BioSapiens project (see `www.biosapiens.info`), we decided that all DAS servers that annotate the UniProt knowledge base should provide the MD5 checksum of the protein sequence that was annotated. This makes it easy for a protein DAS client to determine if the annotations from a certain DAS server are out of date or not.

The way we decided to do this was for each protein DAS server to modify the VERSION attribute of the SEGMENT tag in the XML reply to each DAS request from a client. This probably only makes sense to you if you know something about the XML that makes up a standard DAS feature reply, but ProServer makes it relatively easy to implement, even if you don't.

The `SourceAdaptor` module contains a stub for a method called `segment_version` which receives the name of a segment (the UniProt accession number) as an argument and which is supposed to return the version of the segment as a simple string.

The idea is to let the source adaptor overload the `segment_version` method, and from it return the MD5 checksum of the protein that was used to arrive at the annotation. An overriding implementation of this method in a source adaptor might possibly look something like this (assuming, among other things, that the MD5 checksum has already been calculated and stored in a database):

```
sub segment_version
{
    my ( $self, $segment ) = @_;

    my $dbtable = $self->config()->{'dbtable'};

    my $query = qq(
        SELECT  checksum
        FROM    $dbtable
        WHERE   accession_number = '$segment'
    );

    my $rows = $self->transport()->query($query);

    return $rows->[0]->{'checksum'};
}
```

9. Adding more attributes to the XML response

Another thing we would like to do within the BioSapiens project, since it is such a large project with many diverse data sources each defining its own annotation types, is to introduce a `category_ontology` attribute (which would be a URL pointing to a proper ontology, such as SO or GO, used in the existing `category` attribute), an `evidence` attribute (which would be e.g. a GO evidence code), and a `evidence_ontology` attribute (which would point to the ontology used for the `evidence` attribute). These would all be new attributes in the reply from the `types` request.

Doing this involves changes in two places:

1. in the adaptor that pulls the relevant data from the underlying data source, and
2. in the ProServer code that generates the XML reply.

9.1 Modifying the adaptor

The first of these two changes is the most straight forward. We simply need to add the appropriate keys to the hashes that we send back from the `build_types` method within the adaptor module. Using the example adaptor from earlier in this document, the code that needs changing is this:

```
sub build_types
{
    # Begins as before...

    foreach my $row ( @{$rows} ) {
        push @types,
          {
            'type'   => $row->{'type'},
            'method' => $row->{'method'},
            'count'  => $row->{'count'}
          };
    }

    return @types;
}
```

Now, we change this into

```
sub build_types
{
    # Begins as before...

    foreach my $row ( @{$rows} ) {
        push @types,
          {
            'type'       => $row->{'type'},
            'method'     => $row->{'method'},
            'count'      => $row->{'count'},
            'category'   => $row->{'category'},
            'c_ontology' => 'http://song.sourceforge.net/so.shtml',
            'evidence'   => $row->{'evidence'},
            'e_ontology' =>
              'http://www.geneontology.org/GO.evidence.shtml'
          };
    }

    return @types;
}
```

The only thing we've done here is to add more key-value pairs to the hash. Of course, the query into the database needs to be changed and the data in the database should include the GO evidence code for each type and the SO term for each category as well as the category itself. We're assuming that the ontologies will be constant.

The `category` key is already standard, so we need not do anything about that in our next step, but the ProServer framework does not know about the three new keys that we've used and would ignore them completely and nothing would be gained from adding them here unless we change the `SourceAdaptor` module.

9.2 Modifying ProServer internals

The `SourceAdaptor` module consists of various methods that first of all gets the request from the client (the genome or protein browser) via the `Daemon` module, and then interacts with the adaptor module to create an XML document that gets sent back to the client.

The interesting method in this module is the `das_types` method. It calls the `build_types` method in the adaptor and from the data it receives from there builds the resulting XML document using simple `print` statements.

The part of the method that we will be modifying is the last XML-building step (here beautified a bit, and some long lines has been broken up):

```
    for my $type ( @{ $data->{$seg} } ) {
        $type->{'count'} ||= "";

        my $method = qq(method="$type->{'method'}")
          if ( defined $type->{'method'} );

        my $category = qq(category="$type->{'category'}")
          if ( defined $type->{'category'} );

        $response .=
            qq(<TYPE id="$type->{'type'}")
          . qq( $method $category>$type->{'count'}</TYPE>\n);
    }
    $response .= qq(  </SEGMENT>\n);
```

We just follow the same pattern when adding our new attributes:

```
    for my $type ( @{ $data->{$seg} } ) {
        $type->{'count'} ||= "";

        my $method = qq(method="$type->{'method'}")
          if ( defined $type->{'method'} );

        my $category = qq(category="$type->{'category'}")
          if ( defined $type->{'category'} );

        my $c_ontology = qq(category_ontology="$type->{'c_ontology'}")
          if ( defined $type->{'c_ontology'} );

        my $evidence = qq(evidence="$type->{'evidence'}")
          if ( defined $type->{'evidence'} );

        my $e_ontology = qq(evidence_ontology="$type->{'e_ontology'}")
          if ( defined $type->{'e_ontology'} );

        $response .=
            qq(<TYPE id="$type->{'type'}")
          . qq( $method $category $c_ontology $evidence $e_ontology>)
          . qq($type->{'count'}</TYPE>\n);
    }
    $response .= qq(  </SEGMENT>\n);
```

CONTENTS

Author: Andreas Kähäri, `andreas.kahari@ebi.ac.uk`

Document revision and date:

```
$Revision: 1.20 $
$Date: 2006/03/06 11:49:03 $
```