Tntnet quick start

Authors: Tommi Mäkitalo, Andreas Welchlin

This quick start includes:

- how to install tntnet
- build and run your first application
- explanation of this first application
- further readings

Tntnet is developped and tested with linux. It is known to run on Sun Solaris, IBM AIX and freeBSD.

Installation

For installing Tntnet you need to install cxxtools before.

You find cxxtools on the tntnet homepage http://www.tntnet.org/download.hms and install it with:

```
tar xzf cxxtools-1.4.0.tar.gz
cd cxxtools-1.4.0
./configure
make
su -c 'make install'
```

Same installation-procedure with tntnet. Install it with:

```
tar xzf tntnet-1.5.0.tar.gz
cd tntnet-1.5.0
./configure
make
su -c 'make install'
```

Now you have a working Tntnet-Environment.

How to create your first webapplication

To create a webapplication it is necessary to create some initial projectfiles. This is achieved by entering:

```
tntnet-config --project=myfirstproject
```

This creates:

- a directory “myfirstproject”
- a source-file “myfirstproject.ecpp” containing your application
- a configurationfile “tntnet.conf”
- a property-file for logging-configuration “tntnet.properties”
- a Makefile

To build the application change to the new directory and execute „make“.

To run the application enter „tntnet -c tntnet.conf“.

Now you can start your Webbrowser and navigate to http://localhost.8000/myfirstproject.

You can see the result of your first running tntnet-application, which prints the name of the application.

What have we done?

The sourcefile myfirstproject.ecpp has been translated to c++. This c++ programm was used to build a shared library which contains the whole webapplication.

A tntnet-webapplication is a simple web-page with special tags like **<$** ... **$>**. The ecppc-compiler creates a c++-sourcefile and a headerfile with the same basename. They include a class, which has also the same named as the file. You can look into the generated code, if you want and sometimes it is useful to read it for further understanding of tntnet-applications. If the c++-compiler has problems with your application it is also the best choice to look into the generated code.

Please keep in mind that the linenumbers which are printed by the c++-compiler on errors correspond to the generated cpp-file. They are not the linenumbers of your ecpp-source.

The tags **<$** ... **$>** include a c++-expression. The result of the expression is printed into the resulting page, when the page is requested (on runtime). Therefore a **std::ostream** is used, so that the type of the result can be any object, which has an outputoperator **operator<<(ostream&, T)** defined.

The configurationfile „tntnet.conf“ has 3 configurationvariables. “**Listen**” configures the (local) IP-adress and tcp-port, where tntnet will listen for incoming requests. If no port is configured, the

default is 80, which is normally only possible when tntnet runs with root privileges.

The variable “**PropertyFile**” tells tntnet, where to find the logging-configuration.

The entry “**MapUrl**” is the most important one. It tells tntnet what to do with incoming requests. Without this entry, tntnet answeres every request with *„http-error 404 – not found“*. “**MapUrl**” maps the url - which is sent from a webbrowser - to a tntnet-component. A component is the piece of code, which is normally generated by the ecpp-compiler (*ecppc*).

That's what we did obove with myfirstproject.ecpp. Components are identified by their (class-)name and the shared library which contains this class. We named our class “myfirstproject” and our shared library “myfirstproject.so”. The component-identifier is then myfirstproject@myfirstproject .

So “MapUrl” tells tntnet to call this component, when the url /test.html is requested.

How to add an image to your webapplication

A nice webapplication is colorfull and has some images. Let's add one.

Create or fetch some pictures. Say you have a picture “*picture.jpg”*. Put it into your working-directory.

Modify your html-page “myfirstproject .ecpp” to show an image, first:

```
<html>
 <head>
  <title>ecpp-application myfirstproject</title>
 </head>
 <body>
  <h1>myfirstproject</h1>
  <img src=”picture.jpg”>
 </body>
</html>
```

Next we compile the modified webpage including the picture and link everything together. We need to tell the ecpp-compiler (*ecppc)*, that the picture is a binary file and which mime-type to generate. The flag -b tells ecppc not to look for tags like <$...$>. The component needs to tell the browser the mime-type which is “image/jpg” for your picture. The option **-m** is used to tell ecppc the mime-type. The picture will be compiled into the component.

```
ecppc myfirstproject.ecpp
g++ -c -fPIC myfirstproject.cpp
ecppc -b -m image/jpeg picture.jpg
g++ -c -fPIC picture.cpp
g++ -o myfirstproject.so -shared myfirstproject.o picture.o -lecpp
```

But you can compile this easier by editing the generated Makefile and change the line:

```
myfirstproject.so: myfirstproject.o
```

to:

```
myfirstproject.so: myfirstproject.o picture.o
```

Before tntnet is started it is necessary to extend our configuration. Tntnet needs to know, that "picture.jpg" is found in the shared library "myfirstproject.so". Our new *tntnet.conf* looks like this:

```
MapUrl /myfirstproject.html    myfirstproject@myfirstproject
MapUrl /picture.jpg    picture@test
Listen 0.0.0.0 8000
PropertyFile tntnet.properties
```

Now we start our modified webapplication which is found in myfirstproject.so. Start tntnet like before and look at the modified page including your image.

Generalise the configuration

When adding new pages to tntnet applications you have to ensure, that tntnet finds all the components. Until now we have added each single component into tntnet.conf. There is a way to generalise it by using regular expressions. Just modify tntnet.conf like this:

```
MapUrl /(.*).html     $1@ myfirstproject
MapUrl /(.*).jpg      $1@ myfirstproject
Listen 0.0.0.0 8000
PropertyFile tntnet.properties
```

Every request will be checked by tntnet for matching the first of all regular expressions which are defined. Every request with the suffix ".html" or ".jpg" makes tntnet to look for a component with the basename of the request. Ok – there is one funny thing in our configuration: we get our picture with http://localhost:8000/picture.html. But tntnet does not care and nor does the browser.

Adding some C++-processing

Tntnet is made for writing web applications in C++. In the first example you saw one type of tag: **<$**...**$>**. This encloses a C++-expression, which is evaluated and printed on the resulting page.

Web applications often need to do some processing like fetching data from a database or something. The tags **<{** ... **}>** enclose a C++-processing-block. This C++-code is executed, when a browser sends a request to fetch the page.

As a short form you can put the character '**%**' into the first column, which means, that the rest of the line is C++.

We change our myfirstproject.ecpp to look like that:

```
<html>
 <head>
  <title>ecpp-application myfirstproject</title>
 </head>
 <body>
  <h1>myfirstproject</h1>
  <{
    // we have a c++-block here
```

```
    double arg1 = 1.0;
    double arg2 = 3.0;
    double result = arg1 + arg2
  }>
  <p>
   <$ arg1 $> + <$ arg2 $> =
% if (result == 0.0) {
   nothing
% } else {
  <$ result $>
% }
  </p>
</body>
</html>
```

Compile and run the application with:

```
make
tntnet -c tntnet.conf
```

Maybe we should call it calc.ecpp. Sounds like a better name for a little calculator.

But to be a real calculator the user should be able to enter the values. There is a solution to this, so go on reading.

Processing parameters

Html has forms for dealing with user input. Forms send their values to a web application. The application needs to receive these values as parameters. Tntnet supports this by using the ecpp-tags <**%args**> ... </**%args**> which enclose a parameter definition.

Let's start with a simple example:

```
<%args>
namefield;
</%args>
<html><body>
<form>
What's your name?
<input type=”text” name=”namefield”>
<input type=”submit”>
</form>
<hr>
Hello <$ namefield $>
</body></html>
```

We put a variablename into an args-block. This defines a C++ variable of type **std::string**, which receives the value of the request parameter. The first time we call our application, there are no parameters, so 'namefield' is an empty string.

It is possible to define some an non-empty default value by changing the definition to:

```
<%args>
namefield = “World!”;
</%args>
```

The first time our application is called we get this famous “Hello World!”-output (sorry that it took so long until you get it).

Now we know all instruments which are needed to create a slightly more functional calculator:

```
<%args>
arg1;
arg2;
</%args>
<{
  double v1, v2;
  std::istringstream s1(arg1);
  s1 >> v1;
  std::istringstream s2(arg2);
  s2 >> v2;
}>
<html><body>
<form>
<input type=”text” name=”arg1” value=”<$arg1$>”>
+
<input type=”text” name=”arg2” value=”<$arg2$>”>
% if (s1 && s2) { // if both input-streams were successful extracting values
= <$ v1 + v2 $>
% }
</form>
</body></html>
```

Modularise a web application

A great feature of tntnet is the possibility to create web pages by calling subroutines. You can create small html-snippets and put them together into one big page. First we create a menu, so we create a file with the name menu.ecpp:

```
<a href=”page1.html”>Page 1</a><br>
<a href=”page2.html”>Page 2</a><br>
<a href=”page3.html”>Page 3</a><br>
<a href=”page4.html”>Page 4</a><br>
```

Now we create 4 pages *page1.html* to *page4.html* with some content. We want our menu to be on each of our pages and the following code shows how the menu-component is embedded.

This is page 1:

```
<html>
<body>
<table>
 <tr>
  <td><& “menu” &></td>
  <td><h1>Here is page 1</h1></td>
 </tr>
</table>
</body>
</html>
```

It should not be too hard to derive page 2 to 4 from here. Our Makefile looks like this:

```
OBJECTS=page1.o page2.o page3.o page4.o menu.o
CC=g++
CXXFLAGS=-fPIC

%.cpp %.h: %.ecpp
   ecppc $<

pages.so: $(OBJECTS)
   g++ -o $@ -shared -lecpp $^
```

The configuration does not differ too much from the first example. Just replace @myfirstproject with @pages, because our module name is now pages.so instead of myfirstproject.so.

Call “make” to compile it and as usual run the application with “tntnet -c tntnet.conf”.

A block **<&** ... **&>** contains a subcomponent-call. In our simple case we have a normal C++-string-constant here. It can be also a variable or a functioncall, which returns a std::string.

Further Readings

- tntnet users guide (tntnet.pdf)
- Tntnet configuration-reference (tntnet-configuration.pdf)
- The demo programs in directory “sdk/demos”