Enduro/X Internal/External Developer Guide

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0	2012-12	Initial draft	MV

Contents

1 Enduro/X Development standard **1**

1.1 Reserved identifier prefixes . . . 1

1.2 Global variable naming policy . . . 1

1.3 Reserved UBF field numbers . . . 2

2 Unit testing **3**

3 Enduro/X libraries **4**

4 Common configuration **5**

4.1 Enduro/X common config setup . . . 5

4.2 User accessible configuration server . . . 6

4.3 Common configuration internals . . . 6

5 Common Debug logging API - TPLOG **8**

5.1 Logging facilities . . . 8

5.2 Hierarchy of the loggers (facilities) . . . 8

5.3 Debug string format . . . 9

5.4 Brief of logging functions . . . 9

5.4.1 Part of the standard library (ndebug.h) . . . 9

5.4.2 Part of the XATMI library (xatmi.h) . . . 10

5.5 Request logging concept . . . 10

5.6 Understanding the format of log file . . . 11

6 Queuing mechanisms **13**

7 Object-API **15**

7.1 Class model . . . 17

8 Generating source code with Enduro/X generators 18
8.1 Implementing custom generators . . . 19
8.2 Building sample application generators . . . 19
8.2.1 Prepare project folder3 . . . 19
8.2.2 Generate UBF table for both C & Go . . . 19
8.2.3 Generate C client code & make . . . 22
8.2.4 Generate Go server code & make . . . 23
8.2.5 Provision runtime and put binaries symlinks . . . 23
8.2.6 Run the client . . . 27

9 Using unsolicited messages 29
9.1 Unsolicited message callback processing . . . 29
9.2 Networked operations . . . 30
9.3 Unsolicited message applications . . . 30

10 Adding Enduro/X bindings 31

11 Plugin interface 32
11.1 Plugin Initialization . . . 32
11.2 NDRX_PLUGIN_FUNC_ENCKEY functions . . . 32

12 Additional documentation 34
12.1 Internet resources . . . 34

13 Glossary 35

Chapter 1

Enduro/X Development standard

Enduro/X build system is CMake. Version used should be 2.8 and above. It uses Flex and Bison for building UBF (FML) Expressions.

Intension is done with as 4x spaces. Enduro/X is programmed in NetBeans C/C++ project. NetBeans supports CMake projects.

Project also uses *libxml2* (provided by system), *exhash* (already included in *include* dir) and cgreen (integrated into Enduro/X) for unit testing.

Reserved identifier prefixes

As the C language do not have prefixes like for high level languages (Java, C#, etc), for C developers have to prefix their identifier so that there is no conflict between different party code blocks. This is the case for Enduro/X too. Enduro/X have reserved following keywords as a prefixes for identifiers:

1. NDRX - system wide internal Enduro/X identifiers
2. ndrx - system wide internal Enduro/X identifiers
3. EX - system wide internal Enduro/X identifiers
4. ex - system wide internal Enduro/X identifiers
5. tp - used for user functions for ATMI protocol
6. B - used for UBF buffer API
7. atmi - internal identifiers for tp funcs

Global variable naming policy

Global variables (non static exported from the object file) shall be named with following scheme:

1. *ndrx_G_<variable name>*.

The old naming scheme included only *G* in the front, but we are moving the the common naming scheme with *NDRX*/*ndrx* in the front of the all exported identifiers.

Reserved UBF field numbers

Enduro/X have reserved some list of typed UBF buffer field identifiers for internal use. The list is following:

1. 1-3999
2. 6000-10000
3. 30000001-33554431

For user following field IDs are available:

1. 4000-5999
2. 10001-30000000

Chapter 2

Unit testing

Bot UBF and ATMI sub-systems are unit tested. UBF tests are located under *ubftest* folder, which could be run by:

```
$ ./ubfunit1 2>/dev/null
Running "main"...
Completed "main": 5749 passes, 0 failures, 0 exceptions.
```

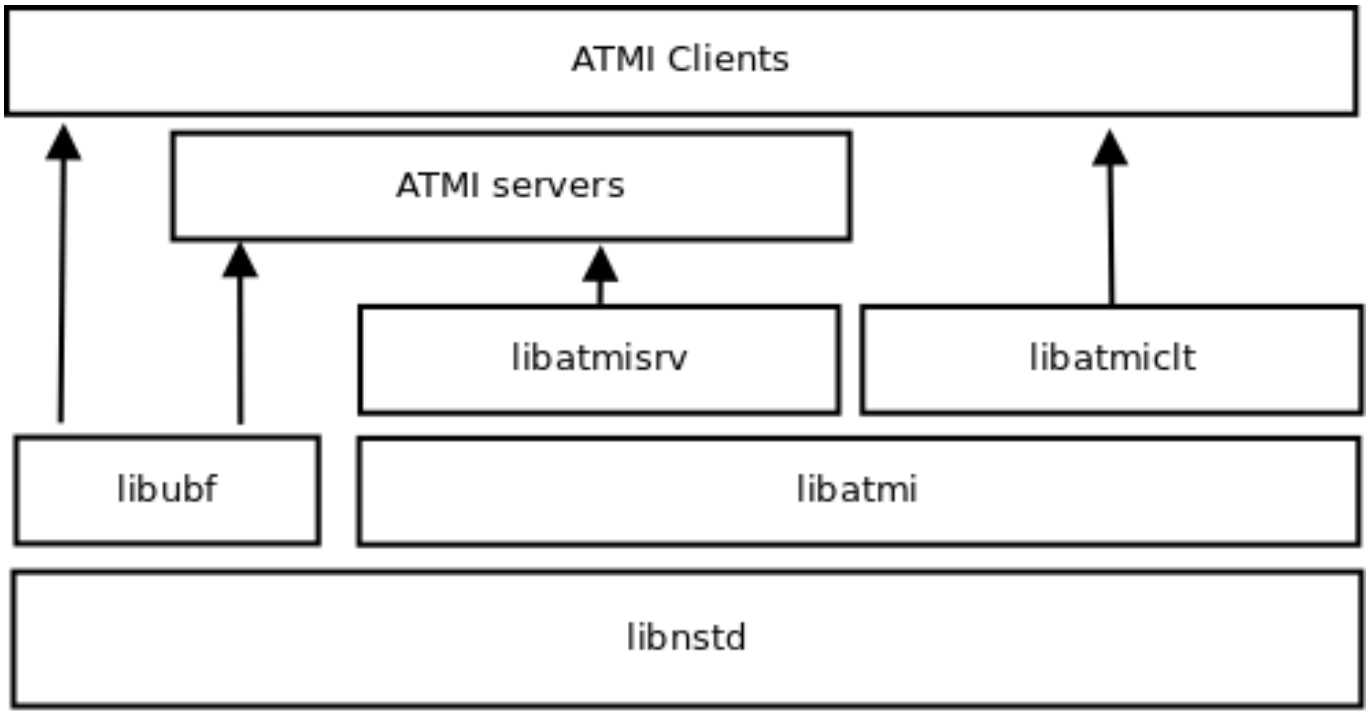
ATMI tests are located at *atmitest* directory, can be run by:

```
$ ./run.sh
tail -n1 test.out
Completed "main": 18 passes, 0 failure, 0 exceptions.
```

Chapter 3

Enduro/X libraries

The framework is composed by following internal libraries and it’s dependencies:



Chapter 4

Common configuration

Enduro/X users are welcome to use common configuration engine. This engine uses ini files to get key/values from ini section (and subsection with inheritance). The configuration can point to directory and in that case Enduro/X will read the all configuration files in directory which ends with with ".ini .cfg, .conf, .config". Configuration engine will automatically detect that given resource is directory and will start to scan for files in directory.

The library keeps all ini file data in memory in hash tables, which also can be iterated as the linked lists. The library can be instructed to refresh the memory configuration. Refresh function detects any files changed in disk (by time stamp) and reload the data in memory.

Enduro/X common config setup

Enduro/X can be configured by using ini file (or files) instead of environment variables, ndrxdebug.conf and q.conf. Two new environment variables now are added to the system:

1. NDRX_CCONFIG=/path/to/ini/file/or/directory/with/files
2. And optional NDRX_CCTAG which allows processes to specify the subsection of Enduro/X system settings.

The configurations sections are:

- [@global] - environment variables for process (see ex_env(5))
- [@debug] - debug configuration per binary (see ndrxdebug.conf(5))
- [@queue] - persistent queue configurations.

If you use NDRX_CCTAG or specify the "cctag" for ATMI server, then Enduro/X will scan for sections like (e.g. cctag=TAG1):

- [@global/TAG1] and then [@global]
- [@debug/TAG1] and then [@debug]
- [@queue/TAG1] and then [@debug]

cctag can contain multiple tags, which are separated by /. In that case multiple lockups will be made with base section combination.

User accessible configuration server

"cconfsrv" XATMI server which can be used by applications to use Enduro/X framework for application configuration. The user application can call the "@CCONFIG" server in two modes:

A) for getting the exact section;

B) for listing the sections.

See cconfsrv(8) for more details.

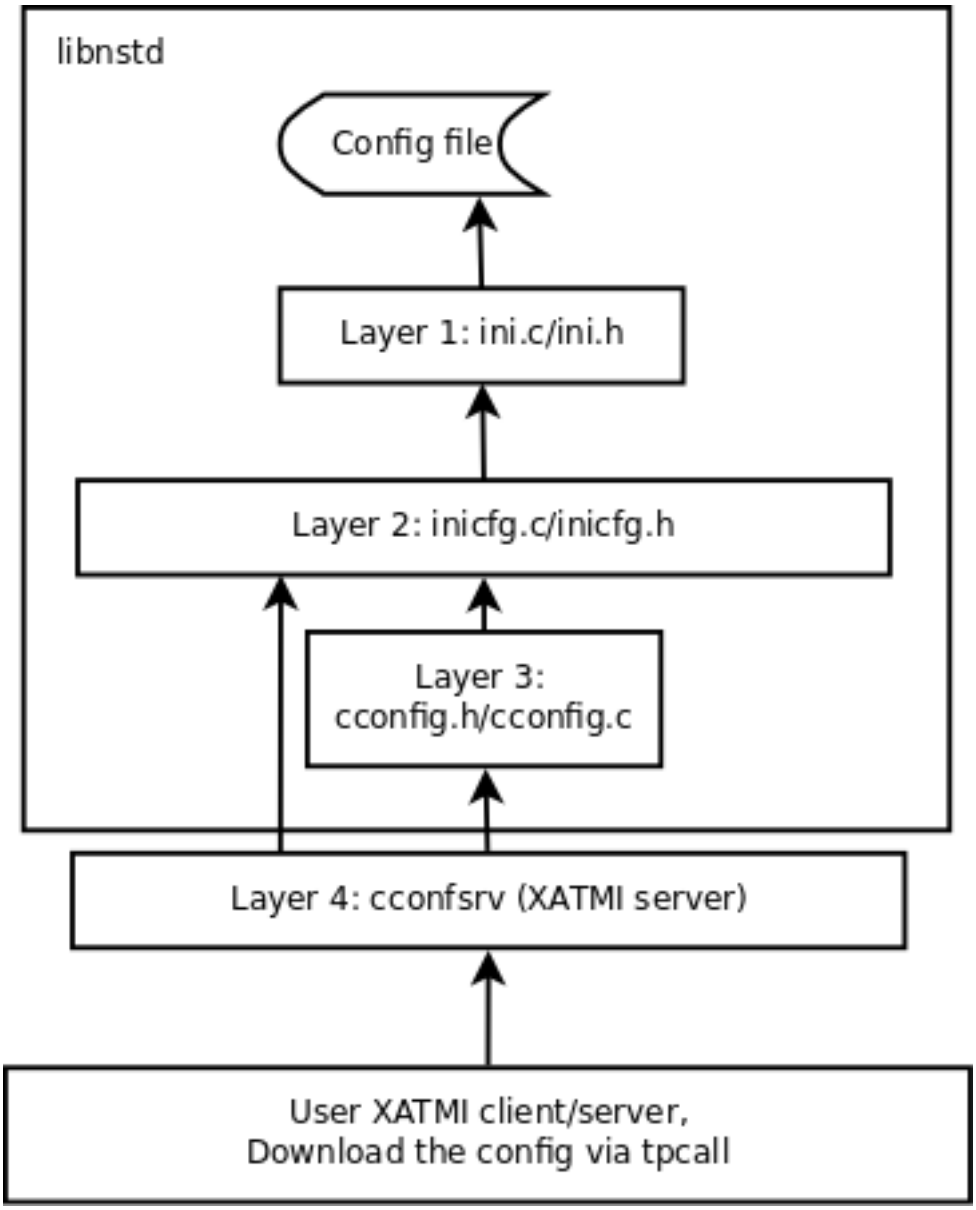
The idea behind this is that user can do the programming under Enduro/X in multiple languages (C/Go/Perl/Python/PHP/NodeJS) and these modules typically needs configuration. It would be waste of time if for each of the languages developer would need to think how to read the configuration from configuration files with native libraries. The Enduro/X offers standard XATMI micro-service call for reading the ini files in common way for whole application, no matter in which language it is programmed.

But C/C++ programmers can use Enduro/X direct libraries for configuration processing. See the *atmitest/test029_inicfg/atmiclt29.c* for sample code.

Common configuration internals

The configuration driving is built in multiple layers:

- Layer 1: Physical file reading by "ini.h" library which gives the callback for any parsed key/value/section;
- Layer 2: Enduro/X code named "inicfg.h" and "inicfg.c". This drives the configuration object loads files into memory. Performs the refreshes, resolves the sections (with inheritance). Returns the buffers with values.
- Layer 3: High level configuration driving by "cconfig.h" and "cconfig.c". This operates with Enduro/X environment variables and Enduro/X configuration files. However you may use different env variables for different purposes. For example: "NDRX_CCONFIG" variable can point to Enduro/X config, but "NDRX_CCONFIG1" can point to your application configuration. And this still is valid setup and keeps files separate.
- Layer 4: "cconfsrv". This is high level API, accessible by transaction protocol (TP) sub-system. See the cconfsrv(8) manpage. Internally is uses Layer 2 and 3 API.


libnstd
Config file
Layer 1: ini.c/ini.h
Layer 2: inicfg.c/inicfg.h
Layer 3:
cconfig.h/cconfig.c
Layer 4: cconfsrv (XATMI server)
User XATMI client/server,
Download the config via tpcall

Chapter 5

Common Debug logging API - TPLOG

Enduro/X offer debug logging facility named "TPLOG". TPLog basically stands for extended user log. The user applications can use this API to configure TPLog, NDRX and UBF logs to be redirect to specified files, configure levels. Enduro/X introduces concept of request logging which means that each system request (or session) which processes UBF buffers can be logged to separate file. Which basically redirects NDRX, UBF and TPLog (user) to specific file. File can be set by **tplogsetreqfile(5)**.

Logging facilities

- *NDRX*, logging facility code *N* - this is Enduro/X XATMI framework internal debug logging. Debug string setting for level is set with keyword *ndrx*. Facility is defined with macros **LOG_FACILITY_NDRX**.
- *UBF*, logging facility code *U* - this is UBF library logs. In debug string level is set with keyword *ubf*. Facility is defined with macros *LOG_FACILITY_UBF*.
- *TP*, logging facility code *t* - this is user logs. In debug string level is set with keyword *tp*. Facility is defined with macros *LOG_CODE_TP*. This is process based logging.
- *TP_THREAD*, logging facility code *T* - this is user logs, set on per thread basis. The log level is set with keyword *tp*. Facility is defined with macros *LOG_FACILITY_TP_THREAD*.
- *TP_REQUEST*, logging facility code *R* - this is user logs, set on per thread/request basis. The log level is set with keyword *tp*. Facility is defined with macros *LOG_FACILITY_TP_REQUEST*.
- *NDRX_THREAD*, logging code *n* - logs the Enduro/X internals on thread basis.
- *UBF_THREAD*, logging code *u* - logs UBF internals on thread basis.
- *NDRX_REQUEST*, logging code *m* - logs the Enduro/X internals on per request basis.
- *UBF_REQUEST*, logging code *v* - logs UBF internals on per request basis.

Hierarchy of the loggers (facilities)

The loggers output the debug content in following order of the facilities status (i.e. definition of current logger):

- If *TP_REQUEST* is open (debug file set), then all logging (TP) will go here. There will be no impact if *TP_REQUEST* log level is different. The request logging can be open by **tplogsetreqfile(3)**. Logger can be closed by **tplogclosereqfile(3)**.
- If *TP_THREAD* is open (debug file set), then all logs of TP will log here. Thread logger can be open by doing **tplogconfig(LOG_FACILITY_TP_THREAD, ...)**. Thread logger can be closed by **tplogclosethread(3)**

- The above principles applies to NDRX_THREAD/REQUEST and UBF_THREAD/REQUEST too.
- NOTE: That that Thread and request logger might have lower or the same log levels as for main loggers. The higher log level than main log level will be ignored.

If there is no TP_REQUEST or TP_THREAD facilities open, then logging is done on per process basis, where there are 3 facilities which are always open:

- *NDRX*, here XATMI sub-system is logged. It can be configured to use separate file by **tplogconfig(3)**.
- *UBF*, here UBF sub-system is logged. It can be configured to use separate file by **tplogconfig(3)**.
- *TP*, here TPLog sub-system is logged. It can be configured to use separate file by **tplogconfig(3)**.

Debug string format

The **debug string** format is described in **ndrxdebug.conf(5)** manpage. basically it is following:

- ndrx=<Debug level> ubf=<Debug level> tp=<Debug level> bufsz=<Number of line to write after doing fflush> file=<log file name, if empty, then stderr>

The debug level is one of the following:

1. No logging output
2. Fatal
3. Error
4. Warning
5. Program info
6. Debug

Brief of logging functions

Enduro/X debugging API offers following list of the functions:

Part of the standard library (ndebug.h)

- void tplogdump(int *lev*, char **comment*, void **ptr*, int *len*); - Dumps the binary buffer (hex-dump) to current logger
- void tplogdumpdiff(int *lev*, char **comment*, void **ptr1*, void **ptr2*, int *len*); - Compares two binary buffers and prints the hex-dump to current logger
- void tplog(int *lev*, char **message*); - Prints the message to current logger, at given log level
- int tploggetreqfile(char **filename*, int *bufsize*); - Get the current request file (see the next chapter)
- int tplogconfig(int *logger*, int *lev*, char **debug_string*, char **module*, char **new_file*); Configure logger. The loggers can be binary *ored* and with one function call multiple loggers can be configured. *lev* is optional, if not set it must be -1. Debug string is optional, but if have one then it can contain all elements. *module* is 4 symbols log module code using in debug lines. *new_file* if set (not NULL and not EOS(0x00)) then it have priority over the file present in debug string.
- void tplogclosereqfile(void); - Close request file. The current logger will fall-back to either thread logger (if configured) or to process loggers.
- void tplogclosethread(void); - Close thread logger, if it was configured.
- void tplogsetreqfile_direct(char *filename); - Set the request file, directly to logger. This operation is used by next function which allows to store the current request logging function in the XATMI UBF buffer.

Part of the XATMI library (xatmi.h)

- int tplogsetreqfile(char ***data*, char ***filename*, char ***filesvc*); - Set the request file. If *data* is UBF buffer allocated by **tpcalloc(3)**, then it will search for *EX_NREQLOGFILE* field presence there. If field present, then *TP_REQUEST* logger will be set to. If field not present, but *filename* is set (not NULL and not EOS), then request logger will be set to this file and name will be loaded into buffer. If file name is not in the buffer and not in the *filename* but *filesvc* present then this XATMI service will be called with *data* buffer and it is expected that field *EX_NREQLOGFILE* will be set which then is used for logging.
- int tploggetbufreqfile(char **data*, char **filename*, int *bufsize*); - Get the request logging file name from XATMI buffer, basically this returns *EX_NREQLOGFILE* value.
- int tplogdelbufreqfile(char **data*); - Delete the request logging information from XATMI buffer.
- void tplogprintubf(int *lev*, char **title*, UBFH **p_ub*); - print the UBF buffer to current logger.

Request logging concept

Request logging is concept when each user session or transaction which is processed by multiple XATMI clients and servers, are logged to single trace file. This is very useful when system have high load with request. Then administrators can identify single transaction and with this request log file it is possible to view full sequence of operation which system performed. You do not need anymore to grep the big log files (based on each service output) and glue together the picture what have happened in system for particular transaction.

The basic use of the request logging is following:

Client process:

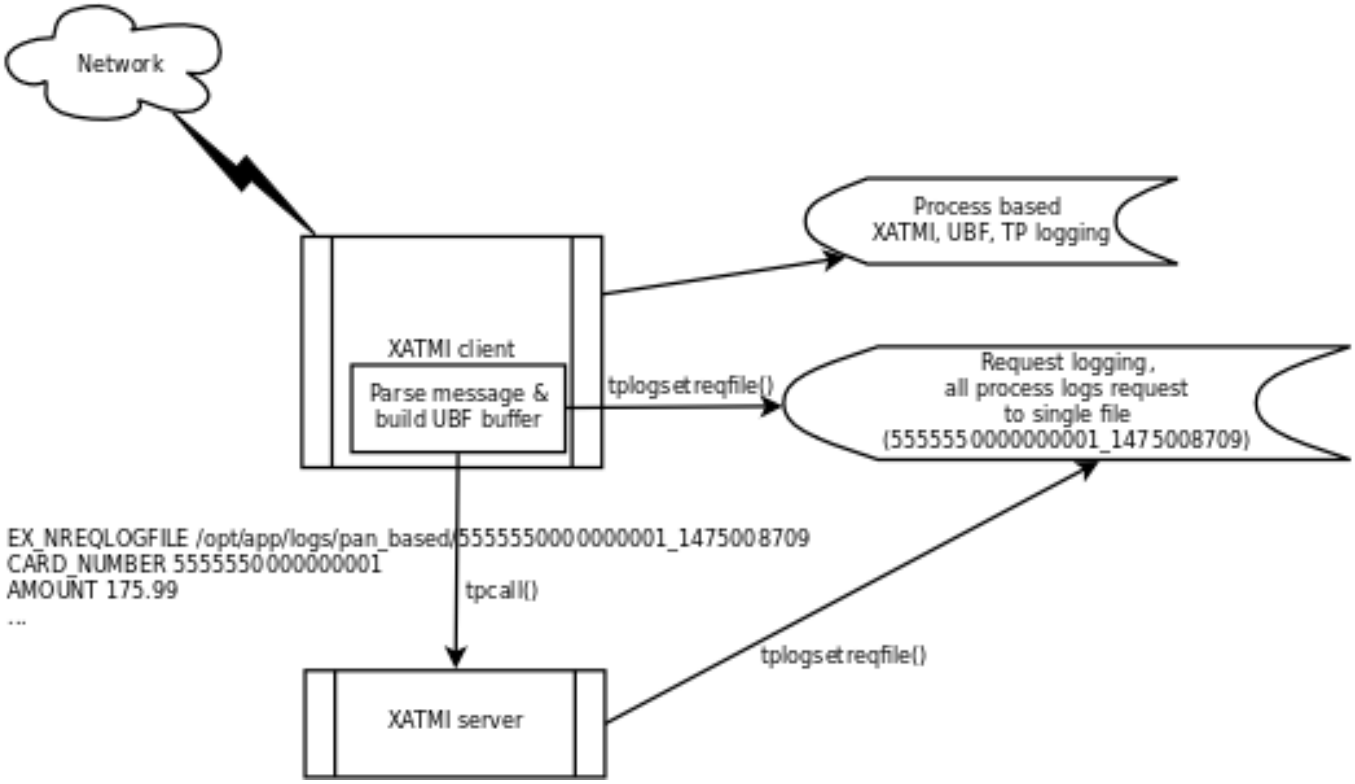
```
/* read the request from network & parse
 * get the transaction subject (for example bank card number (PAN))
 * open the log file for each bank card request
 * e.g.
 */

tplogsetreqfile(&p_ub, "/opt/app/logs/pan_based/<PAN>_<Time_stamp>", NULL);

tplog("About to authorize");

tpcall("AUTHORIZE", &p_ub, ...);

/* reply to network */

tplog("Transaction complete");

/* close the logger after transaction complete */
tplogclosereqfile();
```

Server process - AUTHORIZE service

```
void AUTHORIZE(TPSVCINFO *p_svc)
{
    UBFH *p_ub = (UBFH *)p_svc->data;

    /* Just print the buffer */
    tplogsetreqfile((char **)&p_ub, NULL, NULL);

    tplogprintubf(log_debug, "AUTHORIZE got request", p_ub);

    tplog(log_debug, "Processing...!");

    /* do the work */
```

```
        /* close the request file as we are done. */
    tplogclosereqfile();

    tpreturn(  TPSUCCESS,
                0L,
                (char *)p_ub,
                0L,
                0L);
}
```

Lest assume that for our transaction logfile is set to: */opt/app/logs/pan_based/5555550000000001_1475008709* then transaction could look like:



Understanding the format of log file

For example given code:

```
#include <ndebug.h>

int main (int argc, char **argv)
{
        tplog(5, "Hello from function logger");

        TP_LOG(log_debug, "Hello from macro logger [logging level %d]", log_debug);

        return 0;
}
```

Will print to log file following messages:

```
t:USER:5:11064:000:20160928:100225252:/tplog.c:0412:Hello from function logger
t:USER:5:11064:000:20160928:100225252:ogtest.c:0007:Hello from macro logger [logging level  ↩
    5]
```

So in general log line format is following:

```
<LOGGER_FACILITY>:<MODULE>:<LOG_LEVEL>:<PID>:<OS_THREAD_ID>:<THREAD_ID>:<DATE>:<TIME_MS>:< ↩
    SOURCE_FILE>:<LINE>:<MESSAGE>
```

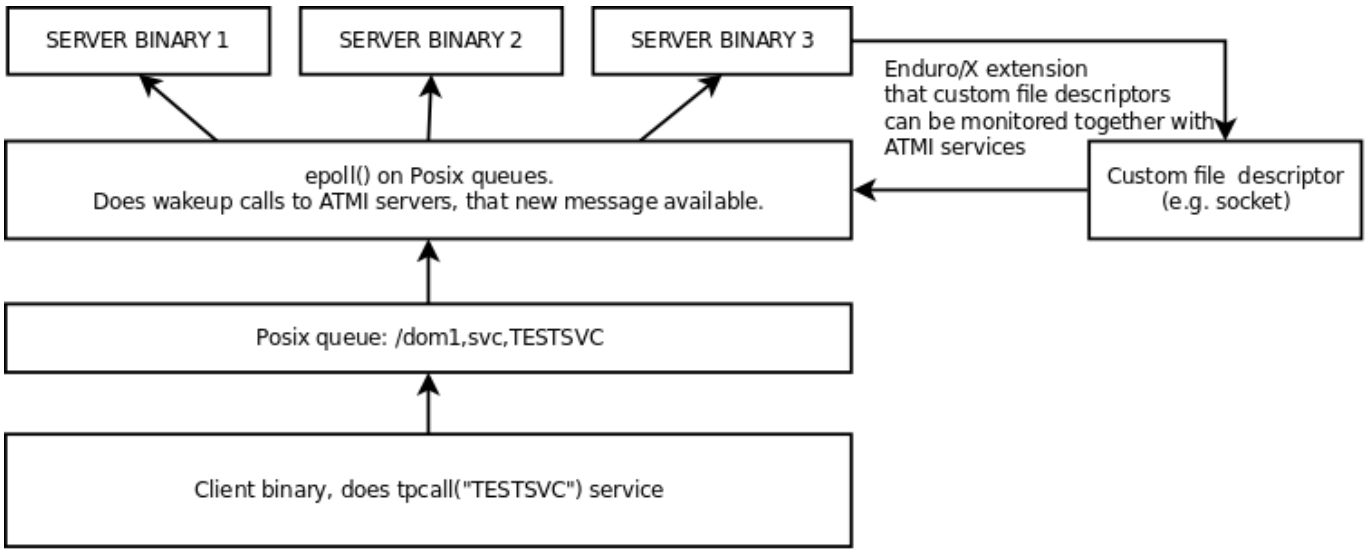
Where:

- *LOGGER_FACILITY* - is logger code which to which message is logged, i.e. *N* - NDRX process based logger, *U* - UBF process based logger, *t* - TP log, process based, *T* - TP thread based logger, *R* - TP request logger, *n* - Enduro/X internals (NDRX) thread logger, *m* - Enduro/X internals (NDRX) request logger, *u* - UBF thread logger, *v* - UBF request logger.
- *MODULE* - 4 char long logger, *NDRX* and *UBF ' or user given code by **tplogconfig(3)**. Default is 'USER*.
- *LOG_LEVEL* - message log level digit.
- *PID* - process id.
- *OS_THREAD_ID* - Operating system thread id (provided by libc or so).
- *THREAD_ID* - internal Enduro/X thread identifier.
- *DATE* - YYYYMMDD time stamp of the message (date part) in local TZ.
- *TIME_MS* - HHmmssSSS - time stamp of the message (time part) in local TZ.
- *SOURCE_FILE* - last 8 symbols of C/C++ source file from which macro logger was called.
- *LINE* - line number of the message in source code (where the macro logger was called).
- *MESSAGE* - logged user message.

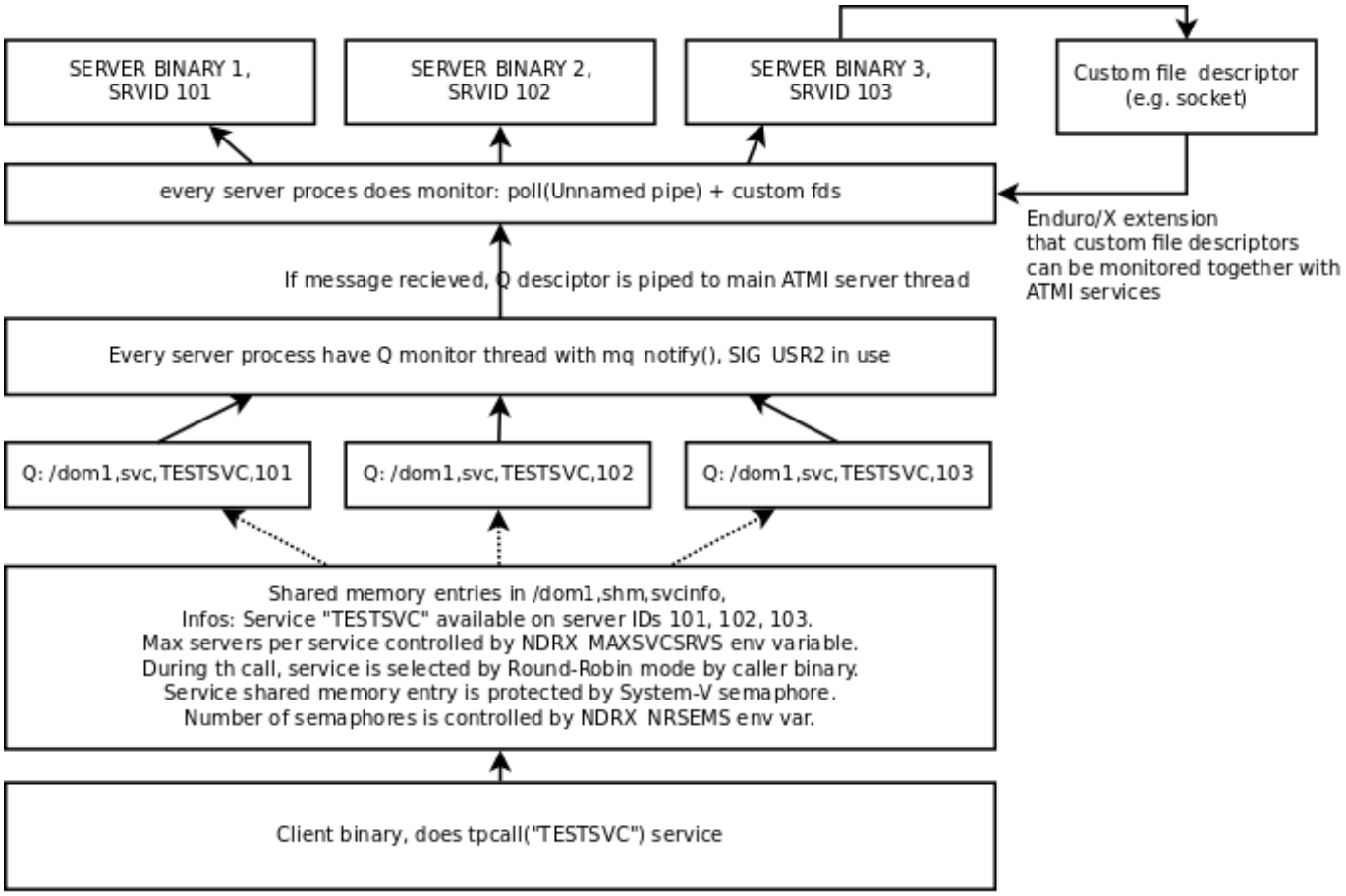
Chapter 6

Queuing mechanisms

Enduro/X originally was developed for GNU/Linux kernels where resource polling sub-system i.e. epoll() can handle Posix queue handlers. This is linux non-standard feature which greatly supports system development. This allows to build one queue - multiple servers architecture (even for ATMI server processes waiting on different queues). However, this this feature limits platform to be working on Linux only.



Starting from Enduro/X version 3, there is support for other Posix compatible Operating Systems. Where possible Posix queues are used. If no Queueu support built in, for example Apple OSX, then emulated Posix queues are used. For these platforms, the caller processes does choose the queue where to send the message in round-robin mode. For each service shared memory contains list of server IDs providing the service. In round robin mode the server id is selected, and caller sends the service to queue (e.g. *dom1,svc,TESTSVC,102* where *102* is server id.).



For other unix support, mq_notify() call for each open queue is installed, by employing SIGUSR2. Signal handling is done in seperate thread. The main ATMI server thread is doing *poll()* in unnamed pipe. When event from mq_ sub-system is received, it writes the queue descriptor id to unnamed pipe and that makes main thread to wake up for queue processing. The *poll()* for main thread supports Enduro/X extensions to add some other resource for polling (e.g. socket fd.)

Chapter 7

Object-API

Enduro/X provides Object API functions. This is meant to be used with integration into programming languages and frameworks, where cooperative multi-threading is used. This API also is suitable for systems like Node.JS where system call, e.g. C lang call can result in different operating system thread. This fact can cause lot of issues, for example, in cooperative multi-threading two concurrent *tpacall()* requests can return results for different cooperative threads, which will cause them to drop the response and both calls with might finish with time-out.

Thus Enduro/X provides following header files for Object-API:

- odebug.h - ATMI Object based debugging
- oubf.h - ATMI Object based UBF operations
- oatmi.h - ATMI operations via ATMI Object
- oatmisrv.h - ATMI server operations via ATMI Object.

The API basically consists of all UBF and ATMI functions, they are prefixed with letter *O* and as first parameter all of them consume *TPCONTEXT_T* typed parameter. Which basically is pointer to heap stored ATMI Object. This ATMI Object also includes links to Standard library and UBF library heap allocated objects.

Every Object-API function basically does following:

1. Set (call of *tpsetctxt()*) the current thread TLS to passed in context;
2. Call the actual UBF/ATMI function;
3. Unset/get (call of *tpsetctxt()*) the thread local data;

During the Enduro/X C library works, it is assumed that is not preemptive for cooperative threads. Thus above scheme will work for every framework that comply with rule (and mostly it does, because it will break the rules of library C/C++ processing).

The typical code for Object API would be following:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include <oatmi.h>
#include <oubf.h>
#include <odebug.h>
#include <Exfields.h>

int main(int argc, char **argv)
{
    int ret = 0;
```

```
    int cd1;
    UBFH *p_ub1;
    long rsplen;
    /* Allocate new context aka Object */
    TPCONTEXT_T ctx1 = tpnewctxt();

    /* Initialise client session */
    if (SUCCEED!=Otpinit(&ctx1, NULL))
    {
        /* print the thread based logs */
        ONDRX_LOG(&ctx1, log_error, "TESTERROR: Failed to Otpinit 1: %s",
                    Otpstrerror(&ctx1, Otperrno(&ctx1)));
        ret = -1;
        goto out;
    }

    /*Do some client based logging */
    ONDRX_LOG(&ctx1, log_always, "Hello from CTX1");

    if (NULL==(p_ub1 = (UBFH *)Otpalloc(&ctx1, "UBF", NULL, 8192)))
    {
        ONDRX_LOG(&ctx1, log_error, "TESTERROR: Failed to Otpalloc ub1: %s",
                    Otpstrerror(&ctx1, Otperrno(&ctx1)));
        ret = -1;
        goto out;
    }

    /* set some buffer value */
    if (SUCCEED!=OCBchg(&ctx1, p_ub1, EX_CC_CMD, 0, "l", 0L, BFLD_STRING))
    {
        ONDRX_LOG(&ctx1,log_error, "TESTERROR: OCBchg() failed %s",
                OBstrerror(&ctx1, OBerror(&ctx1)));
        ret = -1;
        goto out;
   }

    /* call the server */
    if (FAIL==Otpcall(&ctx1, "SOMESVC", (char *)p_ub1, 0L, (char **)&p_ub1, &rsplen, 0L))
    {
        ONDRX_LOG(&ctx1, log_error, "TESTERROR: Failed to Otpcall 1: %s",
                Otpstrerror(&ctx1, Otperrno(&ctx1)));
        ret = -1;
        goto out;
    }

    /* free the buffer */
    Otpfree(&ctx1, (char *)p_ub1);


    /* terminate ATMI client session */
    if (SUCCEED!=Otpterm(&ctx1))
    {
        ONDRX_LOG(&ctx1, log_error, "TESTERROR: Failed to terminate client 1",
                Otpstrerror(&ctx1, Otperrno(&ctx1)));
        ret = -1;
        goto out;
    }

    /* free the NSTD/UBF/ATMI objects */
    tpfreectxt(ctx1);

out:
```

```
    return ret;

}
```

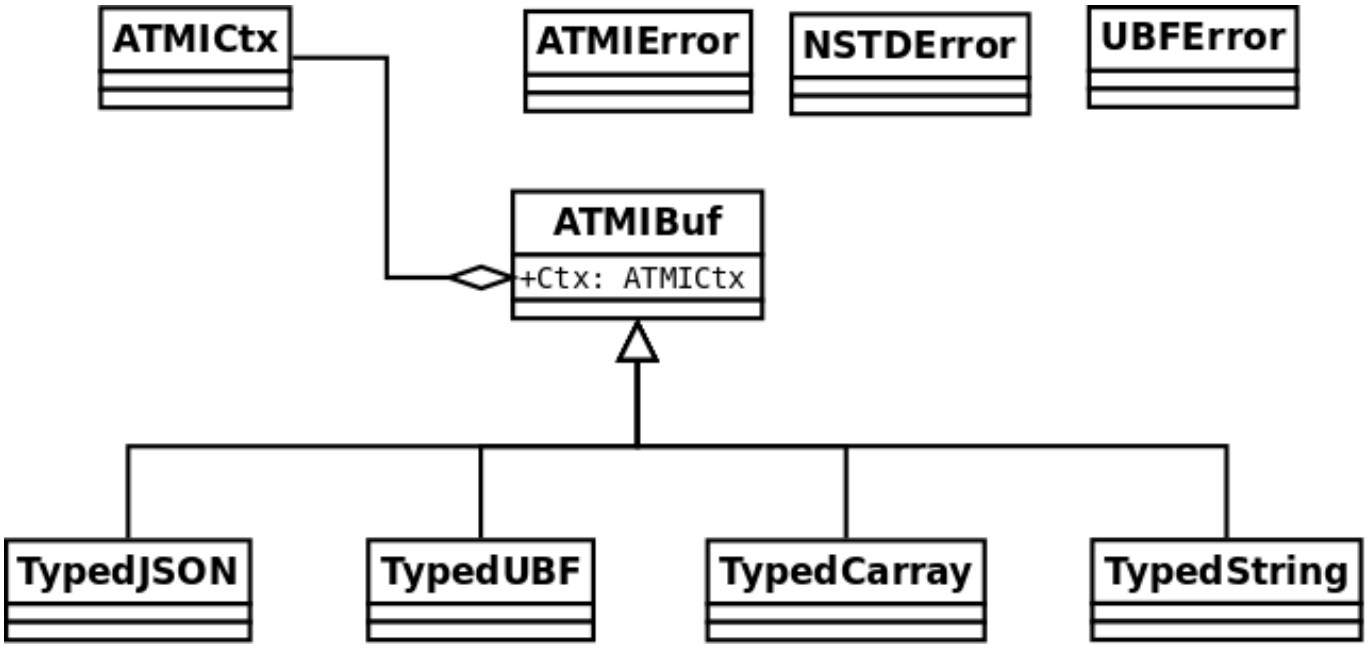
Build with:

```
$ gcc test.c  -latmi -lubf -lnstd -lpthread -lrt -lm -ldl
```

See atmitest/test032_oapi/atmiclt32.c for more sample code.

Class model

For programming languages that supports classes or objects, following class model will be used for Enduro/X bindings.



This diagram is based on *endurox-go* package, which uses structures and special functions that are binded to structure. Basically that is the same as classes.

This model might be implemented for Node.js and Platform Script.

Chapter 8

Generating source code with Enduro/X generators

Enduro/X xadmin command line utility comes with built in generators. Currently following generator targets are available:

- **ubf tab** - Generate *UBF* table header files. This target can generate include file for C, or Go package which constants of the field definitions.
- **c server** - Generate C server. The server can have a common configuration. Wizard offers some options like building a makefile and using a UBF buffer.
- **c client** - Generate C client application. This make sample C client app which in case if UBF buffer is select for data buffer, the sample call is made to **TESTSV** XATMI service.
- **go server** - Go server which depends on **endurox-go** package. Thus in project path the endurox-go package must be installed. (See the sample bellow).
- **go client** - Generate Go XATMI client process. As with Go server, it requires that endurox-go is installed in project path. That can be done by $ go get https://github.com/endurox-dev/endurox-go

The target can be invoked by running $ xadmin gen <target>, for example:

```
$ xadmin gen c server
Enduro/X 3.4.3, build Feb 10 2017 00:34:28, using poll for DARWIN (64 bits)

Enduro/X Middleware Platform for Distributed Transaction Processing
Copyright (C) 2015, 2016 Mavimax, Ltd. All Rights Reserved.

This software is released under one of the following licenses:
GPLv2 (or later) or Mavimax's license for commercial use.

 0: srvname      :XATMI Server Name (binary) [testsv]:
 1: svcnm        :Service name [TESTSV]:
 2: useubf       :Use UBF? [y]: n
 4: genmake      :Gen makefile [y]:

*** Review & edit configuration ***

 0: Edit srvname      :XATMI Server Name (binary) [testsv]:
 1: Edit svcnm        :Service name [TESTSV]:
 2: Edit useubf       :Use UBF? [n]:
 4: Edit genmake      :Gen makefile [y]:
c: Cancel
w: Accept, write
```

```
Enter the choice [0-5, c, w]: w
C server gen ok!

$ make
cc -c -o testsv.o testsv.c -I../ubftab
cc -o testsv  testsv.o -latmisrvinteg -latmi -lubf -lnstd -lpthread -ldl -lm
$
```

Xadmin's package also includes provision scripts which will setup runtime quickly. The command is *$ xadmin provision*.

Implementing custom generators

Enduro/X **xadmin** can be configured with custom generators. The directory or script file name where xadmin looks for Platform Scripts, are configured with following configuration resources:

Building sample application generators

In this section we will make an application where C client code will invoke Go server. The IPC will use UBF buffer, with test fields which are provided by **ubf tab** generator. Also this example assumes that you have installed enduro/x and endurox-go packages to your system and kernel parameters are configured (e.g. queue settings in case of Linux).

Prepare project folder3

Lets assume our project will be made at $TESTHOME. The sources (with sub-projects) will go under $TESTHOME/src. This structure is required for Go projects. For Linux operating system we will set $TESTHOME to **/home/user1/app2**.

```
# useradd -m user1
# su - user1
$ mkdir /home/user1/app2
$ export TESTHOME=/home/user1/app2
$ mkdir $TESTHOME/src
```

Generate UBF table for both C & Go

The application will communicate via Unified Buffer Format (**UBF**) buffer. The test field definitions will be used for this application. Firstly lets generate C headers:

```
$ mkdir $TESTHOME/src/ubftab
$ cd $TESTHOME/src/ubftab

$ xadmin gen ubf tab
Enduro/X 3.4.3, build Feb 10 2017 00:26:22, using epoll for LINUX (64 bits)

Enduro/X Middleware Platform for Distributed Transaction Processing
Copyright (C) 2015, 2016 Mavimax, Ltd. All Rights Reserved.

This software is released under one of the following licenses:
GPLv2 (or later) or Mavimax's license for commercial use.

Logging to ./ULOG.20170211
 0: table_name   :UBF Table name (.fd will be added) [test]:
 1: base_number  :Base number [6000]:
 2: testfields   :Add test fields [y]:
 3: genexfields  :Gen Exfields [y]:
```

```
 4: genmake      :Gen makefile [y]:
 5: makeLang     :Target language (c/go) [c]:

*** Review & edit configuration ***

 0: Edit table_name   :UBF Table name (.fd will be added) [test]:
 1: Edit base_number  :Base number [6000]:
 2: Edit testfields   :Add test fields [y]:
 3: Edit genexfields  :Gen Exfields [y]:
 4: Edit genmake      :Gen makefile [y]:
 5: Edit makeLang     :Target language (c/go) [c]:
c: Cancel
w: Accept, write
Enter the choice [0-6, c, w]: w
Gen ok!

$
```

Now we see that *test.fd.h* is generate. Lets generate Go definitions. Before that we will set *GOPATH* to project root.

```
$ cd $TESTHOME
$ export GOPATH=`pwd`
$ cd $TESTHOME/src/ubftab
$ xadmin gen ubf tab
Enduro/X 3.4.3, build Feb 10 2017 00:26:22, using epoll for LINUX (64 bits)

Enduro/X Middleware Platform for Distributed Transaction Processing
Copyright (C) 2015, 2016 Mavimax, Ltd. All Rights Reserved.

This software is released under one of the following licenses:
GPLv2 (or later) or Mavimax's license for commercial use.

Logging to ./ULOG.20170211
 0: table_name   :UBF Table name (.fd will be added) [test]:
 1: base_number  :Base number [6000]:
 2: testfields   :Add test fields [y]:
 3: genexfields  :Gen Exfields [y]:
 4: genmake      :Gen makefile [y]:
 5: makeLang     :Target language (c/go) [c]: go

*** Review & edit configuration ***

 0: Edit table_name   :UBF Table name (.fd will be added) [test]:
 1: Edit base_number  :Base number [6000]:
 2: Edit testfields   :Add test fields [y]:
 3: Edit genexfields  :Gen Exfields [y]:
 4: Edit genmake      :Gen makefile [y]:
 5: Edit makeLang     :Target language (c/go) [go]:
c: Cancel
w: Accept, write
Enter the choice [0-6, c, w]: w
Gen ok!

$
```

Once the files are generated, we can run off the make:

```
$ cd $TESTHOME/src/ubftab

$ make
make -f Mclang
$SOURCES is [./test.fd Exfields]
```

```
$OUTPUT is [./test.fd.h Exfields.h]
$FIELDTBLS is [./test.fd,Exfields]
make[1]: Entering directory `$TESTHOME/src/ubftab'
mkfldhdr -m0 -pubftab
To control debug output, set debugconfig file path in $NDRX_DEBUG_CONF
N:NDRX:5:  732:2ae627e394c0:000:20170211:163548263:fldhdr.c:0229:Output directory is [.]
N:NDRX:5:  732:2ae627e394c0:000:20170211:163548263:fldhdr.c:0230:Language mode [0]
N:NDRX:5:  732:2ae627e394c0:000:20170211:163548263:fldhdr.c:0231:Private data [ubftab]
N:NDRX:5:  732:2ae627e394c0:000:20170211:163548263:fldhdr.c:0243:Use environment variables
U:UBF :5:  732:2ae627e394c0:000:20170211:163548263:dtable.c:0114:Using NDRX_UBFMAXFLDS:  ↩
    16000
N:NDRX:5:  732:2ae627e394c0:000:20170211:163548263:fldhdr.c:0303:enter generate_files()
U:UBF :5:  732:2ae627e394c0:000:20170211:163548263:fldhdr.c:0138:Load field dir [$TESTHOME/ ↩
    src/ubftab]
U:UBF :5:  732:2ae627e394c0:000:20170211:163548263:fldhdr.c:0149:About to load fields list  ↩
    [./test.fd,Exfields]
N:NDRX:5:  732:2ae627e394c0:000:20170211:163548264:fldhdr.c:0369:$TESTHOME/src/ubftab/./ ↩
    test.fd processed OK, output: ./test.fd.h
N:NDRX:5:  732:2ae627e394c0:000:20170211:163548264:fldhdr.c:0369:$TESTHOME/src/ubftab/ ↩
    Exfields processed OK, output: ./Exfields.h
N:NDRX:5:  732:2ae627e394c0:000:20170211:163548264:fldhdr.c:0256:Finished with : SUCCESS
make[1]: Leaving directory `$TESTHOME/src/ubftab'
make -f Mgolang
$SOURCES is [./test.fd Exfields]
$OUTPUT is [./test.fd.go Exfields.go]
$FIELDTBLS is [./test.fd,Exfields]
make[1]: Entering directory `$TESTHOME/src/ubftab'
mkfldhdr -m1 -pubftab
To control debug output, set debugconfig file path in $NDRX_DEBUG_CONF
N:NDRX:5:  736:2aad91d474c0:000:20170211:163548271:fldhdr.c:0229:Output directory is [.]
N:NDRX:5:  736:2aad91d474c0:000:20170211:163548271:fldhdr.c:0230:Language mode [1]
N:NDRX:5:  736:2aad91d474c0:000:20170211:163548271:fldhdr.c:0231:Private data [ubftab]
N:NDRX:5:  736:2aad91d474c0:000:20170211:163548271:fldhdr.c:0243:Use environment variables
U:UBF :5:  736:2aad91d474c0:000:20170211:163548271:dtable.c:0114:Using NDRX_UBFMAXFLDS:  ↩
    16000
N:NDRX:5:  736:2aad91d474c0:000:20170211:163548271:fldhdr.c:0303:enter generate_files()
U:UBF :5:  736:2aad91d474c0:000:20170211:163548271:fldhdr.c:0138:Load field dir [$TESTHOME/ ↩
    src/ubftab]
U:UBF :5:  736:2aad91d474c0:000:20170211:163548271:fldhdr.c:0149:About to load fields list  ↩
    [./test.fd,Exfields]
N:NDRX:5:  736:2aad91d474c0:000:20170211:163548271:fldhdr.c:0369:$TESTHOME/src/ubftab/./ ↩
    test.fd processed OK, output: ./test.fd.go
N:NDRX:5:  736:2aad91d474c0:000:20170211:163548271:fldhdr.c:0369:$TESTHOME/src/ubftab/ ↩
    Exfields processed OK, output: ./Exfields.go
N:NDRX:5:  736:2aad91d474c0:000:20170211:163548271:fldhdr.c:0256:Finished with : SUCCESS
go build  -o ubftab *.go
go install  ./...
make[1]: Leaving directory `$TESTHOME/src/ubftab'

$ ls -l
total 72
-rw-rw-r-- 1 user1 user1  9641 feb 11 16:25 Exfields
-rw-rw-r-- 1 user1 user1  6079 feb 11 16:35 Exfields.go
-rw-rw-r-- 1 user1 user1  7614 feb 11 16:35 Exfields.h
-rw-rw-r-- 1 user1 user1   145 feb 11 16:25 Makefile
-rw-rw-r-- 1 user1 user1   492 feb 11 16:25 Mclang
-rw-rw-r-- 1 user1 user1   562 feb 11 16:27 Mgolang
-rw-rw-r-- 1 user1 user1  1301 feb 11 16:25 test.fd
-rw-rw-r-- 1 user1 user1  1532 feb 11 16:35 test.fd.go
-rw-rw-r-- 1 user1 user1  1999 feb 11 16:35 test.fd.h
-rw-rw-r-- 1 user1 user1  2882 feb 11 16:35 ubftab
-rw-rw-r-- 1 user1 user1 15464 feb 11 16:27 ULOG.20170211
```

```
$ head -n10 test.fd.h
#ifndef __TEST_FD
#define __TEST_FD
/*      fname   bfldid            */
/*      -----   -----            */
#define T_CHAR_FLD      ((BFLDID32)67114875)    /* number: 6011  type: char */
#define T_CHAR_2_FLD    ((BFLDID32)67114876)    /* number: 6012  type: char */
#define T_SHORT_FLD     ((BFLDID32)6021)        /* number: 6021  type: short */
#define T_SHORT_2_FLD   ((BFLDID32)6022)        /* number: 6022  type: short */
#define T_LONG_FLD      ((BFLDID32)33560463)    /* number: 6031  type: long */
#define T_LONG_2_FLD    ((BFLDID32)33560464)    /* number: 6032  type: long */
```

So it have installed a *ubftab* package, and generated *test.fd.h* file.

Generate C client code & make

Now lets generate a C client code which will send the UBF buffer to Go server. The generator provides C sample client, let's use it.

```
$ mkdir $TESTHOME/src/clt
$ cd $TESTHOME/src/clt

$ xadmin gen c client
Enduro/X 3.4.3, build Feb 10 2017 00:26:22, using epoll for LINUX (64 bits)

Enduro/X Middleware Platform for Distributed Transaction Processing
Copyright (C) 2015, 2016 Mavimax, Ltd. All Rights Reserved.

This software is released under one of the following licenses:
GPLv2 (or later) or Mavimax's license for commercial use.

Logging to ./ULOG.20170211
 0: cltname      :XATMI Client Name (binary) [testcl]:
 1: useubf       :Use UBF? [y]:
 2: ubfname      :UBF include folder name (will be done ../<name>) [ubftab]:
 3: genmake      :Gen makefile [y]:
 4: config       :INI File section (optional, will read config if set) []:

*** Review & edit configuration ***

 0: Edit cltname      :XATMI Client Name (binary) [testcl]:
 1: Edit useubf       :Use UBF? [y]:
 2: Edit ubfname      :UBF include folder name (will be done ../<name>) [ubftab]:
 3: Edit genmake      :Gen makefile [y]:
 4: Edit config       :INI File section (optional, will read config if set) []:
c: Cancel
w: Accept, write
Enter the choice [0-4, c, w]: w
C client gen ok!


$ make
cc -c -o testcl.o testcl.c -I../ubftab
cc -o testcl  testcl.o -latmiclt -latmi -lubf -lnstd -lpthread -lrt -ldl -lm
```

C Client have been generated OK and built ok.

Generate Go server code & make

Now lets generate Go server. Before we make the Go app, we need to get the **endurox-go** package.

```
$ cd $TESTHOME
$ go get github.com/endurox-dev/endurox-go
$ mkdir $TESTHOME/src/srv
$ cd $TESTHOME/src/srv
$ xadmin gen go server
Enduro/X 3.4.4, build Feb 11 2017 16:57:21, using epoll for LINUX (64 bits)

Enduro/X Middleware Platform for Distributed Transaction Processing
Copyright (C) 2015, 2016 Mavimax, Ltd. All Rights Reserved.

This software is released under one of the following licenses:
GPLv2 (or later) or Mavimax's license for commercial use.

Logging to ./ULOG.20170211
 0: svname       :XATMI Server Name (binary) [testsv]:
 1: svcname      :Service name [TESTSV]:
 2: useubf       :Use UBF? [y]:
 3: ubfname      :UBF package name [ubftab]:
 4: genmake      :Gen makefile [y]:
 5: config       :INI File section (optional, will read config if set) []:

*** Review & edit configuration ***

 0: Edit svname       :XATMI Server Name (binary) [testsv]:
 1: Edit svcname      :Service name [TESTSV]:
 2: Edit useubf       :Use UBF? [y]:
 3: Edit ubfname      :UBF package name [ubftab]:
 4: Edit genmake      :Gen makefile [y]:
 5: Edit config       :INI File section (optional, will read config if set) []:
c: Cancel
w: Accept, write
Enter the choice [0-5, c, w]: w
Go server gen ok!


$ make
go build  -o testsv *.go
```

As we see test server was built ok. Now next step is to configure a runtime system. With provisioning of the configuration files and adding testsv to boot application boot sequence.

Provision runtime and put binaries symlinks

To create a runtime system, we will use $ xadmin provision command. This command allows to register one server to ndrxconfig.xml. For demo application purposes this is fully fine. The provision will be done in root directly of "bankapp2".

```
$ cd $TESTHOME


$ ls -l
total 8
drwxrwxr-x 3 user1 user1 4096 feb 11 16:27 pkg
drwxrwxr-x 8 user1 user1 4096 feb 11 17:05 src


$ xadmin provision
Enduro/X 3.4.4, build Feb 11 2017 16:57:21, using epoll for LINUX (64 bits)
```

```
Enduro/X Middleware Platform for Distributed Transaction Processing
Copyright (C) 2015, 2016 Mavimax, Ltd. All Rights Reserved.

This software is released under one of the following licenses:
GPLv2 (or later) or Mavimax's license for commercial use.

Logging to ./ULOG.20170212


    _______              __                      ___  __
   / ____/___  ____  __/ /_  ____________    _/_/ |/ /
  / __/ / __ \/ __  / / / / ___/ __ \ _/_/ |   /
 / /___/ / / / /_/ / /_/ / /  / /_/ //_/  /   |
/_____/_/ /_/\__,_/\__,_/_/   \____/_/   /_/|_|

                     Provision

Compiled system type....: LINUX

 0: qpath        :Queue device path [/dev/mqueue]:
 1: nodeid       :Cluster node id [1]:
 2: qprefix      :System code (prefix/setfile name, etc) [test1]: app2
 3: timeout      :System wide tpcall() timeout, seconds [90]:
 4: appHome      :Application home [$TESTHOME]:
 6: binDir       :Executables/binaries sub-folder of Apphome [bin]:
 8: confDir      :Configuration sub-folder of Apphome [conf]:
 9: logDir       :Log sub-folder of Apphome [log]:
10: ubfDir       :Unified Buffer Format (UBF) field defs sub-folder of Apphome [ubftab]:
11: tempDir      :Temp sub-dir (used for pid file) [tmp]:
12: installQ     :Configure persistent queue [y]:
13: tmDir        :Transaction Manager Logs sub-folder of Apphome [tmlogs]:
14: qdata        :Queue data sub-folder of Apphone [qdata]:
15: qSpace       :Persistent queue space namme [SAMPLESPACE]:
16: qName        :Sample persistent queue name [TESTQ1]:
17: qSvc         :Target service for automatic queue for sample Q [TESTSVC1]:
18: eventSv      :Install event server [y]:
19: cpmSv        :Configure Client Process Monitor Server [y]:
20: configSv     :Install Configuration server [y]:
21: bridge       :Install bridge connection [y]:
22: bridgeRole   :Bridge -> Role: Active(a) or passive(p)? [a]:
24: ipc          :Bridge -> IP: Connect to [172.0.0.1]:
25: port         :Bridge -> IP: Port number [21003]:
26: otherNodeId  :Other cluster node id [2]:
27: ipckey       :IPC Key used for System V semaphores [44000]:
28: ldbal        :Load balance over cluster [0]:
29: ndrxlev      :Logging: ATMI sub-system log level 5 - highest (debug), 0 - minimum (off) ↩
     [5]:2
30: ubflev       :Logging: UBF sub-system log level 5 - highest (debug), 0 - minimum (off)  ↩
    [1]:
31: tplev        :Logging: /user sub-system log level 5 - highest (debug), 0 - minimum (off ↩
    ) [5]:
32: usv1         :Configure User server #1 [n]: y
33: usv1_name    :User server #1: binary name []: testsv
34: usv1_min     :User server #1: min [1]:
35: usv1_max     :User server #1: max [1]:
36: usv1_srvid   :User server #1: srvid [2000]:
37: usv1_cctag   :User server #1: cctag []:
38: usv1_sysopt  :User server #1: sysopt []:
Invalid value: Min length 1
38: usv1_sysopt  :User server #1: sysopt []: -e ${NDRX_APPHOME}/log/testsv.log
39: usv1_appopt  :User server #1: appopt []:
50: ucl1         :Configure User client #1 [n]:
```

```
55: addubf       :Additional UBFTAB files (comma seperated), can be empty []: test.fd
56: msgsizemax   :Max IPC message size [56000]:
57: msgmax       :Max IPC messages in queue [100]:

*** Review & edit configuration ***

 0: Edit qpath        :Queue device path [/dev/mqueue]:
 1: Edit nodeid       :Cluster node id [1]:
 2: Edit qprefix      :System code (prefix/setfile name, etc) [app2]:
 3: Edit timeout      :System wide tpcall() timeout, seconds [90]:
 4: Edit appHome      :Application home [$TESTHOME]:
 6: Edit binDir       :Executables/binaries sub-folder of Apphome [bin]:
 8: Edit confDir      :Configuration sub-folder of Apphome [conf]:
 9: Edit logDir       :Log sub-folder of Apphome [log]:
10: Edit ubfDir       :Unified Buffer Format (UBF) field defs sub-folder of Apphome [ubftab ↩
    ]:
11: Edit tempDir      :Temp sub-dir (used for pid file) [tmp]:
12: Edit installQ     :Configure persistent queue [y]:
13: Edit tmDir        :Transaction Manager Logs sub-folder of Apphome [tmlogs]:
14: Edit qdata        :Queue data sub-folder of Apphone [qdata]:
15: Edit qSpace       :Persistent queue space namme [SAMPLESPACE]:
16: Edit qName        :Sample persistent queue name [TESTQ1]:
17: Edit qSvc         :Target service for automatic queue for sample Q [TESTSVC1]:
18: Edit eventSv      :Install event server [y]:
19: Edit cpmSv        :Configure Client Process Monitor Server [y]:
20: Edit configSv     :Install Configuration server [y]:
21: Edit bridge       :Install bridge connection [y]:
22: Edit bridgeRole   :Bridge -> Role: Active(a) or passive(p)? [a]:
24: Edit ipc          :Bridge -> IP: Connect to [172.0.0.1]:
25: Edit port         :Bridge -> IP: Port number [21003]:
26: Edit otherNodeId  :Other cluster node id [2]:
27: Edit ipckey       :IPC Key used for System V semaphores [44000]:
28: Edit ldbal        :Load balance over cluster [0]:
29: Edit ndrxlev      :Logging: ATMI sub-system log level 5 - highest (debug), 0 - minimum  ↩
    (off) [2]:
30: Edit ubflev       :Logging: UBF sub-system log level 5 - highest (debug), 0 - minimum ( ↩
    off) [1]:
31: Edit tplev        :Logging: /user sub-system log level 5 - highest (debug), 0 - minimum ↩
     (off) [5]:
32: Edit usv1         :Configure User server #1 [y]:
33: Edit usv1_name    :User server #1: binary name [testsv]:
34: Edit usv1_min     :User server #1: min [1]:
35: Edit usv1_max     :User server #1: max [1]:
36: Edit usv1_srvid   :User server #1: srvid [2000]:
37: Edit usv1_cctag   :User server #1: cctag []:
38: Edit usv1_sysopt  :User server #1: sysopt [-e ${NDRX_APPHOME}/log/testsv.log]:
39: Edit usv1_appopt  :User server #1: appopt []:
50: Edit ucl1         :Configure User client #1 [n]:
55: Edit addubf       :Additional UBFTAB files (comma seperated), can be empty [test.fd]:
56: Edit msgsizemax   :Max IPC message size [56000]:
57: Edit msgmax       :Max IPC messages in queue [100]:
c: Cancel
w: Accept, write
Enter the choice [0-57, c, w]: w
ndrxconfig: [$TESTHOME/conf/ndrxconfig.xml]
appini: [$TESTHOME/conf/app.ini]
setfile: [$TESTHOME/conf/setapp2]


To start your system, run following commands:
$ cd $TESTHOME/conf
$ source setapp2
```

```
$ xadmin start -y


Provision succeed!

$ ls -l
total 68
drwxrwxr-x 2 user1 user1  4096 feb 12 10:32 bin
drwxrwxr-x 2 user1 user1  4096 feb 12 10:32 conf
drwxrwxr-x 2 user1 user1  4096 feb 12 10:32 log
drwxrwxr-x 3 user1 user1  4096 feb 11 16:27 pkg
drwxrwxr-x 2 user1 user1  4096 feb 12 10:32 qdata
drwxrwxr-x 8 user1 user1  4096 feb 11 17:05 src
drwxrwxr-x 3 user1 user1  4096 feb 12 10:32 tmlogs
drwxrwxr-x 2 user1 user1  4096 feb 12 10:32 tmp
drwxrwxr-x 2 user1 user1  4096 feb 12 10:32 ubftab
-rw-rw-r-- 1 user1 user1 30755 feb 12 10:32 ULOG.20170212
```

Once the system is provisioned, we need to put the symbolic links to our binaries to Enduro/X runtime "bin" directory. Also we will put our test field definition file **test.fd** into **$TESTHOME/ubftab** folder.

```
$ cd $TESTHOME/bin

$ ln -s $TESTHOME/src/clt/testcl .

$ ln -s $TESTHOME/src/srv/testsv .

$ cd $TESTHOME/ubftab

$ ln -s $TESTHOME/src/ubftab/test.fd .
```

Now we are ready to boot up the runtime:

```
$ cd $TESTHOME/conf
$ source setapp2
$ xadmin start -y
Enduro/X 3.4.4, build Feb 11 2017 16:57:21, using epoll for LINUX (64 bits)

Enduro/X Middleware Platform for Distributed Transaction Processing
Copyright (C) 2015, 2016 Mavimax, Ltd. All Rights Reserved.

This software is released under one of the following licenses:
GPLv2 (or later) or Mavimax's license for commercial use.

EnduroX back-end (ndrxd) is not running
ndrxd PID (from PID file): 18849
ndrxd idle instance started.
exec cconfsrv -k 0myWI5nu -i 1 -e $TESTHOME/log/cconfsrv.log -r --  :
        process id=18851 ... Started.
exec cconfsrv -k 0myWI5nu -i 2 -e $TESTHOME/log/cconfsrv.log -r --  :
        process id=18852 ... Started.
exec tpevsrv -k 0myWI5nu -i 20 -e $TESTHOME/log/tpevsrv.log -r --  :
        process id=18853 ... Started.
exec tpevsrv -k 0myWI5nu -i 21 -e $TESTHOME/log/tpevsrv.log -r --  :
        process id=18854 ... Started.
exec tmsrv -k 0myWI5nu -i 40 -e $TESTHOME/log/tmsrv-rm1.log -r -- -t1 -l$TESTHOME/tmlogs/ ↩
    rm1 --  :
        process id=18855 ... Started.
exec tmsrv -k 0myWI5nu -i 41 -e $TESTHOME/log/tmsrv-rm1.log -r -- -t1 -l$TESTHOME/tmlogs/ ↩
    rm1 --  :
        process id=18867 ... Started.
```

```
exec tmsrv -k 0myWI5nu -i 42 -e $TESTHOME/log/tmsrv-rm1.log -r -- -t1 -l$TESTHOME/tmlogs/ ↩
    rm1 --  :
        process id=18879 ... Started.
exec tmqueue -k 0myWI5nu -i 60 -e $TESTHOME/log/tmqueue-rm1.log -r -- -m SAMPLESPACE -s1 -- ↩
      :
        process id=18891 ... Started.
exec tpbridge -k 0myWI5nu -i 150 -e $TESTHOME/log/tpbridge_2.log -r -- -f -n2 -r -i  ↩
    172.0.0.1 -p 21003 -tA -z30 :
        process id=18923 ... Started.
exec testsv -k 0myWI5nu -i 2000 -e $TESTHOME/log/testsv.log --  :
        process id=18924 ... Started.
exec cpmsrv -k 0myWI5nu -i 9999 -e $TESTHOME/log/cpmsrv.log -r -- -k3 -i1 --  :
        process id=18929 ... Started.
Startup finished. 11 processes started.
```

Now test availability of our test service:

```
$ xadmin psc
Enduro/X 3.4.4, build Feb 11 2017 16:57:21, using epoll for LINUX (64 bits)

Enduro/X Middleware Platform for Distributed Transaction Processing
Copyright (C) 2015, 2016 Mavimax, Ltd. All Rights Reserved.

This software is released under one of the following licenses:
GPLv2 (or later) or Mavimax's license for commercial use.

ndrxd PID (from PID file): 6119
Nd Service Name Routine Name Prog Name SRVID #SUCC #FAIL MAX      LAST     STAT
-- ------------ ------------ --------- ----- ----- ----- -------- -------- -----
1  @CCONF       CCONF        cconfsrv  1     0     0     0ms      0ms      AVAIL
1  @CCONF       CCONF        cconfsrv  2     0     0     0ms      0ms      AVAIL
1  @TPEVSUBS    TPEVSUBS     tpevsrv   20    0     0     0ms      0ms      AVAIL
1  @TPEVUNSUBS  TPEVUNSUBS   tpevsrv   20    0     0     0ms      0ms      AVAIL
1  @TPEVPOST    TPEVPOST     tpevsrv   20    0     0     0ms      0ms      AVAIL
1  @TPEVDOPOST  TPEVDOPOST   tpevsrv   20    0     0     0ms      0ms      AVAIL
1  @TPEVSUBS    TPEVSUBS     tpevsrv   21    0     0     0ms      0ms      AVAIL
1  @TPEVUNSUBS  TPEVUNSUBS   tpevsrv   21    0     0     0ms      0ms      AVAIL
1  @TPEVPOST    TPEVPOST     tpevsrv   21    0     0     0ms      0ms      AVAIL
1  @TPEVDOPOST  TPEVDOPOST   tpevsrv   21    0     0     0ms      0ms      AVAIL
1  @TM-1        TPTMSRV      tmsrv     40    0     0     0ms      0ms      AVAIL
1  @TM-1-1      TPTMSRV      tmsrv     40    0     0     0ms      0ms      AVAIL
1  @TM-1-1-40   TPTMSRV      tmsrv     40    0     0     0ms      0ms      AVAIL
1  @TM-1        TPTMSRV      tmsrv     41    0     0     0ms      0ms      AVAIL
1  @TM-1-1      TPTMSRV      tmsrv     41    0     0     0ms      0ms      AVAIL
1  @TM-1-1-41   TPTMSRV      tmsrv     41    0     0     0ms      0ms      AVAIL
1  @TM-1        TPTMSRV      tmsrv     42    0     0     0ms      0ms      AVAIL
1  @TM-1-1      TPTMSRV      tmsrv     42    0     0     0ms      0ms      AVAIL
1  @TM-1-1-42   TPTMSRV      tmsrv     42    0     0     0ms      0ms      AVAIL
1  @TMQ-1-60    TMQUEUE      tmqueue   60    0     0     0ms      0ms      AVAIL
1  @QSPSAMPLES+ TMQUEUE      tmqueue   60    0     0     0ms      0ms      AVAIL
1  @TPBRIDGE002 TPBRIDGE     tpbridge  150   0     0     0ms      0ms      AVAIL
1  TESTSV       TESTSV       testsv    2000  0     0     0ms      0ms      AVAIL
1  @CPMSVC      CPMSVC       cpmsrv    9999  0     0     0ms      0ms      AVAIL
```

TESTSV is advertised, thus all is ok. No try will run the test client.

Run the client

We will run the client by simply invoking in shell **testcl** binary. The working progress will be logged on output.

```
$ testcl
t:USER:4: 6845:7fd1d85b47c0:000:20170212:191211999:testcl.c:0044:Initializing...
t:USER:4: 6845:7fd1d85b47c0:000:20170212:191212000:testcl.c:0090:Processing...
T_STRING_FLD    Hello world!
T_STRING_2_FLD  Hello World from XATMI server
t:USER:4: 6845:7fd1d85b47c0:000:20170212:191212004:testcl.c:0129:Got response from server:  ↩
    [Hello World from XATMI server]
t:USER:4: 6845:7fd1d85b47c0:000:20170212:191212004:testcl.c:0069:Uninitializing...
```

Thus as we see from the sample run it did call the server and got back the response "Hello World from XATMI server". Thus we can conclude that server and client was successfully generated and runtime provisioned.

Chapter 9

Using unsolicited messages

Enduro/X supports unsolicited messages. The idea is that server process (or other client processes which have a handler to client) can send unsolicited messages to clients. The client processes consumes these messages and invokes the callback function. The callback is invoked in case if callback handler is set by **tpsetunsol(3)** function.

The unsolicited messages are posted by XATMI services by using **tpnotify(3)**. This function gets the Client ID (extracted from service call parameter structure, field TPSVCINFO.cltid:

```
void SOMESERVICE (TPSVCINFO *p_svc)
{
    ...
    if (0!=tpnotify(&p_svc->cltid, (char *)p_ub, 0L, 0L))
    {
        NDRX_LOG(log_error, "Failed to tpnotify()!");
        ...
    }
    ...

}
```

Unsolicited messages can be broadcast to client processes by servers and client by using **tpbroadcast(3)**. The broadcast takes Enduro/X cluster node id (*lmid* param) and client name (*cltname* param). The match of the client processes are made by either field present (exact match), field not present (match all) or match by regular expression.

Function signatures are following:

```
int tpnotify(CLIENTID *clientid, char *data, long len, long flags);
int tpbroadcast(char *lmid, char *usrname, char *cltname, char *data, long len, long flags) ↩
    ;
```

Unsolicited message callback processing

The callback function receives XATMI buffer which was provided to the **tpnotify(3)** or **tpbroadcast(3)**. When callback processes these messages, there is limited availability of the operations that can be performed within the callback. The limitation is due to fact, that unsolicited messages are provided from internals of the XATMI runtime and for example doing **tpcall(3)** might cause recursive invocation of the callback handler and can cause stack overflow. The following list of XATMI functions are available during the callback processing:

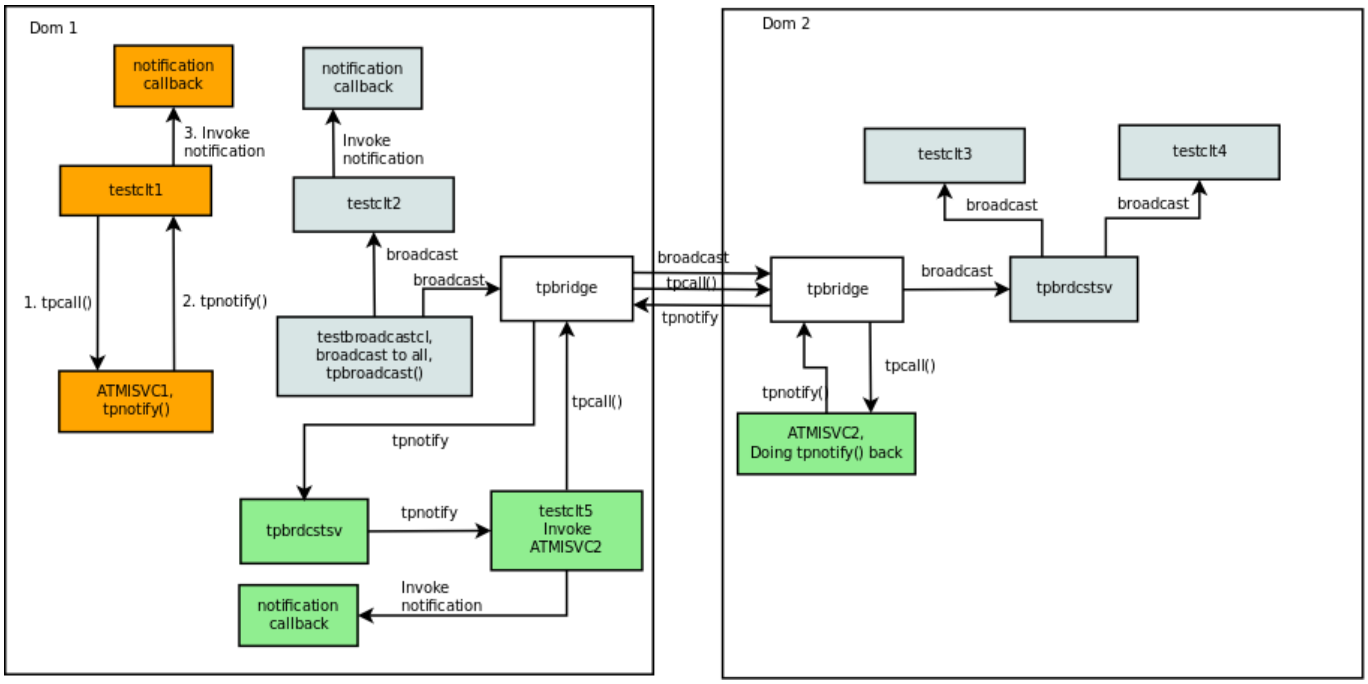
1. **tpalloc(3)**
2. **tpfree(3)**
3. **tpgetlev(3)**

4. **tprealloc(3)**
5. **tptypes(3)**

If more advanced processing is required, the user might create a new thread, copy the XATMI buffer and pass it to the thread. Copy of the buffer is required due to fact, that buffer is automatic made free when callback function returns.

Networked operations

When sending the message to the client to different Enduro/X cluster node, then the transport of the notification is performed by **tpbridge(8)** bridge process, but remote dispatching is performed by special XATMI server named **tpbrdcstsv(8)**. To overall notifications are processed in this way:



the picture contains:

1. Local tpnotify() - orange color
2. Local and remote tpbroadcast() - gray color
3. Remote tpnotify() - green

Unsolicited message applications

Unsolicited messages can be used for XATMI service reporting back progress of some particular work the client. Thus the **tpcall(3)** is not interrupted, but some feedback can be received and processed.

Sample usage can be seen in Enduro/X ATMI test cases 38 and 39.

Chapter 10

Adding Enduro/X bindings

Currently Enduro/X have tier 1 bindings for th Go language. This implementation can be used as reference for other language implementations. The core for the binding development is following:

We classically start with "data structures and algorithms"! Thust firstly define a structures.

But before we start the development, we need to create a build system for target language. The package name is endurox-<language name>, .e.g endurox-java. The build system shall build the corresponding library and test executables.

1. Add enumeration of Enduro/X constants
2. Define error object, either it is just struct or exception classes
3. Create ATMI Context struct/class
4. Define Generic ATMI Buffer Object, add inherited objects to STRING, UBF, JSON, RAW/CARRAY
5. Advertise service (this means from high level language call Ondrx_main(), which will make init callback). Needs to advertise service and allow the ndrx_main() to start to poll for messages. Once the message arrives we need to callback a language specific function.

The bindings will use all libs server & client (like a Go). Thus it depends on the application logic either the binary becomes server or it will be just a client.

Chapter 11

Plugin interface

Enduro/X provides API for writing custom plugins (loaded by shared libraries). There are certain criteria to which plugins must correspond. This chapter will provide the plugin API definition. Also it will list the functionality which can be defined by plugin. Plugins shall be written in thread safe manner.

Plugin Initialization

Plugins are registered in *NDRX_PLUGINS* environment variable, as semicolon separated values. Plugins are loaded during the process "boostarp" (basically at the time when Enduro/X debug logger is initialized, before the Common-Configuration is read. Thus plugins cannot be registered in *[@global]* section. As they must be already loaded before the INI file parsing, as for example custom cryptography provider might be used. Libraries must be available in current shared library search path (e.g. LD_LIBRARY_PATH, DYLD_LIBRARY_PATH, etc..).

Sample configuration:

```
$ export NDRX_PLUGINS=customcrypto.so;somotherfunc.so
```

Enduro/X plugin interface requires two **mandatory** symbols to be exported from plugin library, which must correspond to the following signature:

```
long ndrx_plugin_init(char *provider_name, int provider_name_bufsz);
```

Where *provider_name* is arbitrary string describing the plugin. *provider_name_bufsz* is buffer size for the plugin description. Typically it is around ~60 bytes.

In case of error function shall return **-1**. In case of success init function shall return one or more NDRX_PLUGIN_FUNC_XXXX OR'ed bits, denoting the functionality which is being exported.

Currently following flags are available:

1. NDRX_PLUGIN_FUNC_ENCKEY - plugin provides cryptography key function

During the Initialization, only early logging (mem buffered logs) are available, see NDRX_LOG_EARLY/UBF_LOG_EARLY/TP_LOG
If use of other log functions is made, then must probably program will deadlock.

NDRX_PLUGIN_FUNC_ENCKEY functions

If plugin exports this flag, then library loader will search for following symbol in the shared library:

```
int ndrx_plugin_crypto_getkey(char *keybuf, int keybuf_bufsz);
```

Where *keybuf* is buffer where to install encryption key. The encryption key must be zero (0x00) terminated C string. *keybuf_bufsz* denotes the max buffer size (with 0x00 byte). In case of success function shall return 0. In case of failure, function shall return -1. For this function only **EARLY** logging is available (NDRX_LOG_EARLY/UBF_LOG_EARLY/TP_LOG_EARLY).

numbered!:

Chapter 12

Additional documentation

Internet resources

[1] [ATMI-API] http://docs.oracle.com/cd/E13203_01/tuxedo/tux71/html/pgint6.htm

[2] [FML-API] http://docs.oracle.com/cd/E13203_01/tuxedo/tux91/fml/index.htm

Chapter 13

Glossary

This section lists

ATMI
: Application Transaction Monitor Interface

UBF
: Unified Buffer Format it is similar API as Tuxedo's FML