

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	C Programming style	1
1.1.1	Indentation	1
1.1.2	Variable compare with constants	2
1.1.3	Error handling	2
1.1.4	Code with	3
1.1.5	Function visibility	3
1.1.6	Code documentation	4
1.2	Reserved identifier prefixes	5
1.3	Global variable naming policy	5
1.4	Reserved UBF field numbers	5
2	Unit testing	7
3	Source code indexing with glimpse	8
3.1	Glimpse installation	8
3.2	Source indexing and searching	8
4	Enduro/X libraries	10
5	Common configuration	11
5.1	Enduro/X common configuration setup	11
5.2	User accessible configuration server	12
5.3	Common configuration internals	12
6	Common Debug logging API - TPLOG	14
6.1	Logging facilities	14
6.2	Hierarchy of the loggers (facilities)	14
6.3	Debug string format	15
6.4	Brief of logging functions	15
6.4.1	Part of the standard library (ndebug.h)	15
6.4.2	Part of the XATMI library (xatmi.h)	16
6.5	Request logging concept	16
6.6	Understanding the format of log file	17

7	Queuing mechanisms	19
7.1	Linux epoll and FreeBSD kqueue	19
7.2	System V mode	19
7.3	Poll mode	21
8	Object-API	23
8.1	Class model	25
9	Generating source code with Enduro/X generators	26
9.1	Implementing custom generators	27
9.2	Building sample application generators	27
9.2.1	Prepare project folder3	27
9.2.2	Generate UBF table for both C & Go	27
9.2.3	Generate C client code & make	30
9.2.4	Generate Go server code & make	31
9.2.5	Provision runtime and put binaries symlinks	31
9.2.6	Run the client	35
10	Using unsolicited messages	37
10.1	Unsolicited message callback processing	37
10.2	Networked operations	38
10.3	Unsolicited message applications	38
11	Adding Enduro/X bindings	39
12	Plugin interface	40
12.1	Plugin Initialization	40
12.2	NDRX_PLUGIN_FUNC_ENCKEY functions	41
12.3	NDRX_PLUGIN_FUNC_TPLOGPRINTUBF_HOOK functions	41
13	Starting Enduro/X XATMI server from other thread than main	42
13.1	Enduro/X Process naming strategies	44
13.1.1	Strategy 1	45
13.1.2	Strategy 2	45
13.2	Booting processes as XATMI servers with out CLOPT	45
14	Process forking	46
15	Source code management	47
16	Process debugging	48
16.1	Tracking down memory usage with Valgrind for XATMI servers	48

17 Two Phase Commit Processing notes	49
17.1 Normal Enduro/X Operations of Two Phase Commit	49
17.2 Non XA Switch ready database	49
17.3 TMSRV Transaction log format	49
17.3.1 General info about the transaction	50
17.3.2 Transaction stage identification	50
17.3.3 Resource manager status information	51
18 Additional documentation	53
18.1 Internet resources	53
19 Glossary	54

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.

Enduro/X is programmed in NetBeans C/C++ project. NetBeans supports CMake projects. Some command line editing usually is done by using VIM editor. The source code is appended with mode line settings. Thus to get the proper indentation enabled, configure VIM editor before doing any edits append ~/.vimrc with following commands:

```
set nocompatible
filetype plugin on
set modeline
```

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

1.1 C Programming style

This chapter points out key aspects of doing code changes in Enduro/X C core.

1.1.1 Indentation

For code formatting Enduro/X C code uses Allman Indentation style, e.g.

```
void something(void)
{
    ...
}

while (x == y)
{
    something();
    somethingelse();
}

finalthing();
```

with indention of 4 spaces.

1.1.2 Variable compare with constants

In case if in C compare variable with constants, the constant must be first and then follow the variable. This allows to escape from the errors when accidentally by writing `single = assignment` is made instead of compare, which might lead to unpredictable errors.

For example

```
#define SOME_CONST 10
int i=5;

/* THIS IS BAD STYLE */
if (i==SOME_CONST)
{
    ...
}

/* THIS IS GOOD STYLE */
if (SOME_CONST==i)
{
    ...
}
```

1.1.3 Error handling

All API calls which Enduro/X uses, must be error checked. When error is detected, it must be logged with corresponding logger. Either *ndrx* or *ubf* logger. *ubf* logger (UBF_LOG() macros) shall be used only for libubf library. All other libraries shall log with NDRX_LOG() macros.

The critical errors which are more like system error or some very wrong configuration shall be logged with **userlog()** too. This ensures that during the production operations, administrators can see the

Enduro/X uses "goto" as escape from function in case of error. The all functions that shall handle the errors (i.e. and not return any other error identifier like NULL pointer), the function shall bring with it self an *ret* (return state) variable the simple integer. The *ret* by default shall be set to *EXSUCCEED* from *ndrstandard.h* header. The exit of function shall be marked with "out" label. After the out some de-init and status printing could be done. When error occurs, recommended way to escape is to write some log about the situation and use macro EXFAIL_OUT(*ret*), which effectively set the *ret* to EXFAIL and performs *goto out*;

Some examples of error handling:

```
#include <ndrstandard.h>

...

/**
 * Some error function
 * @param some_arg cannot be less than 0
 * @return EXFAIL (on failure), EXSUCCEED (all OK)
 */
expublic int ndrx_some_func(int some_arg)
{
    int ret = EXSUCCEED;

    if (some_arg < 0)
    {
        /* what is ret=EXFAIL; goto out; */
        EXFAIL_OUT(ret);
    }

out:
```

```
    return ret;
}
```

1.1.4 Code with

The code line length should be more or less 80 symbols. If function arguments are going wider, then moved some to next line. If string goes over the 80 chars then string shall be spitted with standard compiler concatenation done by C.

Typical word wrap should look like:

```
/* lock call descriptor */
if (!(flags & TPNOREPLY))
{
    /* get the call descriptor */
    if (EXFAIL==(tpcall_cd = get_call_descriptor_and_lock(&call->callseq,
        timestamp, flags)))
    {
        NDRX_LOG(log_error, "Do not have resources for "
            "track this call!");
        ndr_x_TPset_error_fmt(TPELIMIT, "%s:All call descriptor entries have been used "
            "(check why they do not free up? Maybe need to "
            "use tpcancel())?", __func__);
        EXFAIL_OUT(ret);
    }
}
else
{
    NDRX_LOG(log_warn, "TPNOREPLY => cd=0");
    tpcall_cd = 0;
}
```

1.1.5 Function visibility

Function visibility in C is controlled by **static** prefix for functions. If one is present, then function visibility is at object file level, if prefix is not present, then function visibility is global exported symbol. To make it more clear, Enduro/X SDK includes two macros

1. `expublic` - empty macros to indicate that function names is available globally.
2. `exprivate` - substitute for **static** keyword. Function visibility is at file level.

for example:

```
#include <ndrstandard.h>

expublic void ndr_x_some_global_func(void)
{
    return;
}

exprivate void some_local_func(void)
{
    return;
}
```


1.1.6 Code documentation

ALL written code must be properly commented, so that other source maintainers can clearly understand what's going on at particular case. Comments are welcome.

Regarding the mandatory documentation, Enduro/X uses Doxygen / JavaDoc style comments for functions, macros and files, so that the API documentation can be generated by doxygen. All function arguments must be documented at any level.

File beginning must start with following block:

```
/**
 * @brief Short description of the file purpose
 *
 * @file file.name
 */
>>> License block <<<
```

The structure for C code is denoted by following comments (see bellow). This also include the sample type definitions and comments for given resources. Where possible grouping of comments shall be made. So that it could be denoted to user commons of the resources.

```
/*-----Includes-----*/
#include <ndrstandard.h> /* Enduro/X standard header */
/*-----Externs-----*/
/** This is global variable */
extern int ndrx_G_some_global; /** < this is other way to document... */
/*-----Macros-----*/
#define HELLO /**< This is hello world macros */

/**
 * This is group of worlds (this is detailed description of group)
 * @defgroup worldsgrp Group of worlds (short description of group)
 * @{
 */

#define WORLD_1 /**< Hello world 1 */
#define WORLD_2 /**< Hello world 2 */
/** Hello world 3 define, use this if the comment line is too large to fit
 * together in 80 symbols
 */
#define WORLD_3
/** @} */ /* end of worldsgrp */

/*-----Enums-----*/
/*-----Typedefs-----*/

/**
 * Some structure used for ...
 */
struct some_struct
{
    int some_field; /**< some field descr */
};

/**
 * This is type of \ref some_struct used for ...
 * thus we reference the structure here.
 */
typedef struct some_struct some_struct_t;

/*-----Globals-----*/
/*-----Statics-----*/
```

```
/*-----Prototypes-----*/
```

Where each of the section shall included the given type of resources declared.

Function documentation is following, by example:

```
/**
 * This is example of some function. This is description of func.
 * @param[in] arg1 this is input argument
 * @param[in,out] arg2 this is output argument
 * @return in case of success function returns ptr to updated \p arg2
 * in case of error NULL is returned.
 */
exp public char *ndrx_some_func(int arg1, char *arg2)
{
    return NULL;
}
```

1.2 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 functions
8. edb - LMDB renamed for Enduro/X internal usage

1.3 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.

1.4 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

Source code indexing with glimpse

So that developers would be more simple to orient in the source code from command line, Enduro/X build system offers use of glimpse tool to index the source code.

3.1 Glimpse installation

On Ubuntu like GNU/Linux systems, glimpse can be installed in following way:

```
$ sudo apt install glimpse
```

On other systems where glimpse does not come out of the box, it can be compiled from source code, download here: <http://webglimpse.net/download.php>

For example:

```
$ wget http://webglimpse.net/trial/glimpse-latest.tar.gz
$ tar -xzf glimpse-latest.tar.gz
$ cd glimpse-4.18.6
$ ./configure
$ make
$ sudo make install
```

Glimpse requires (when compiled from sources) Flex shared library, on Ubuntu this can be installed by:

```
$ sudo apt-get install libfl-dev
```

3.2 Source indexing and searching

Once Enduro/X project is checked out, built and Glimpse is installed, you may index the source code using following make target:

```
$ make index
```

```
This is glimpseindex version 4.18.7, 2015.
```

```
Indexing "/home/user1/projects/endurox" ...
```

```
Size of files being indexed = 9941954 B, Total #of files = 1664
```

```
Index-directory: "/home/user1/projects/endurox/glimpse_index"
Glimpse-files created here:
```

```

-rw-rw-r-- 1 user1 user1    171 Aug 18 07:30 .glimpse_exclude
-rw----- 1 user1 user1 123657 Aug 18 08:59 .glimpse_filenames
-rw----- 1 user1 user1   6656 Aug 18 08:59 .glimpse_filenames_index
-rw----- 1 user1 user1     0 Aug 18 08:59 .glimpse_filetimes
-rw----- 1 user1 user1 451169 Aug 18 08:59 .glimpse_index
-rw----- 1 user1 user1    306 Aug 18 08:59 .glimpse_messages
-rw----- 1 user1 user1    836 Aug 18 08:59 .glimpse_partitions
-rw----- 1 user1 user1 380242 Aug 18 08:59 .glimpse_statistics
Built target index

```

This also generates search command script at project root. So for example, now to search for tpcall, we can use following command from project root (or any other folder, because "/home/user1/projects/endurox/glim" includes full path to project).

```

$ ./glim tpcall
Your query may search about 33% of the total space! Continue? (y/n)y
/home/user1/projects/endurox/tpevsrv/tpevsrv.c:          if (EXFAIL==(tpcallex ( ←
    tmpsvc, p_svc->data, p_svc->len,
/home/user1/projects/endurox/tpevsrv/tpevsrv.c: * Event name carried in extradata of ←
    tpcallex())
/home/user1/projects/endurox/libnetproto/proto.c: #define TTC          7 /* tpcall */
/home/user1/projects/endurox/libatmisrv/tpreturn.c: *                  or tpcall wrapper)
...

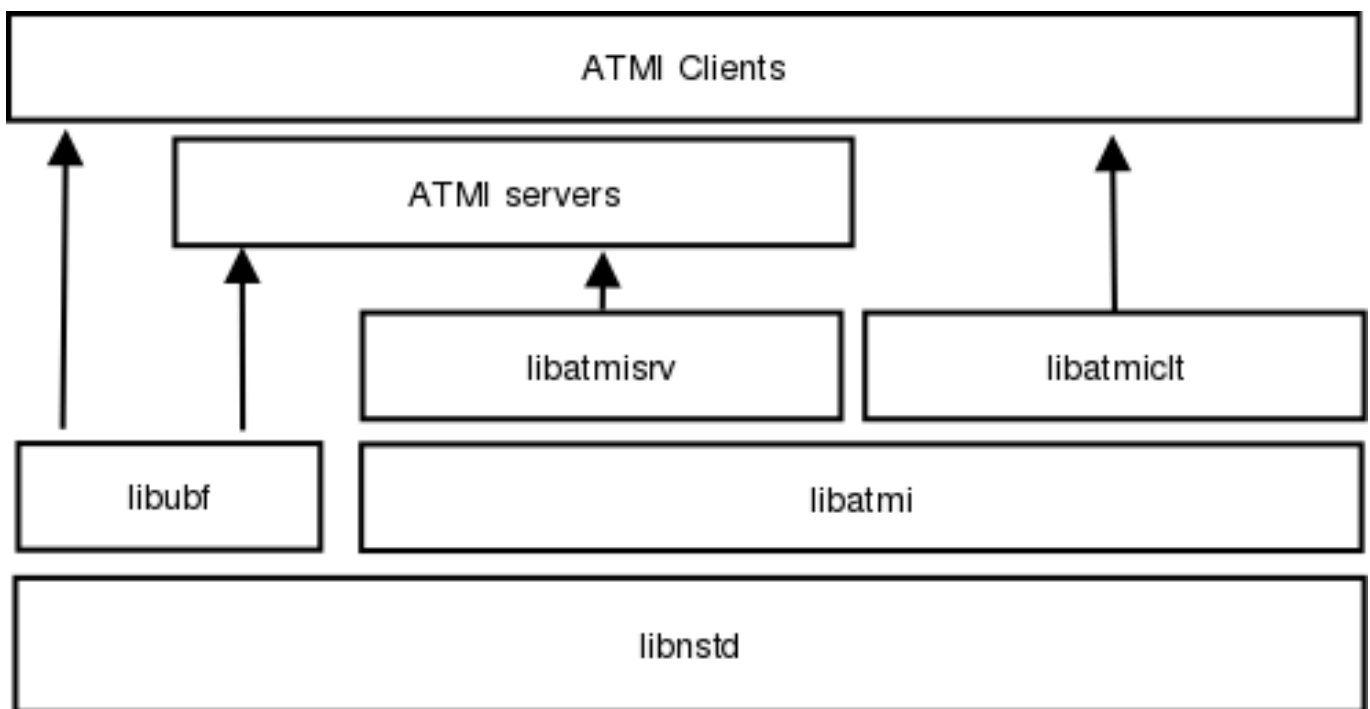
```

In case if some files or directories must be excluded from the index path (used at make index phase), the editing can be done in `glimpse_index/.glimpse_exclude` file at project root directory.

Chapter 4

Enduro/X libraries

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



Chapter 5

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.

5.1 Enduro/X common configuration setup

Enduro/X can be configured by using ini file (or files) instead of environment variables, ndrxddebug.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 ndrxddebug.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.

5.2 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.

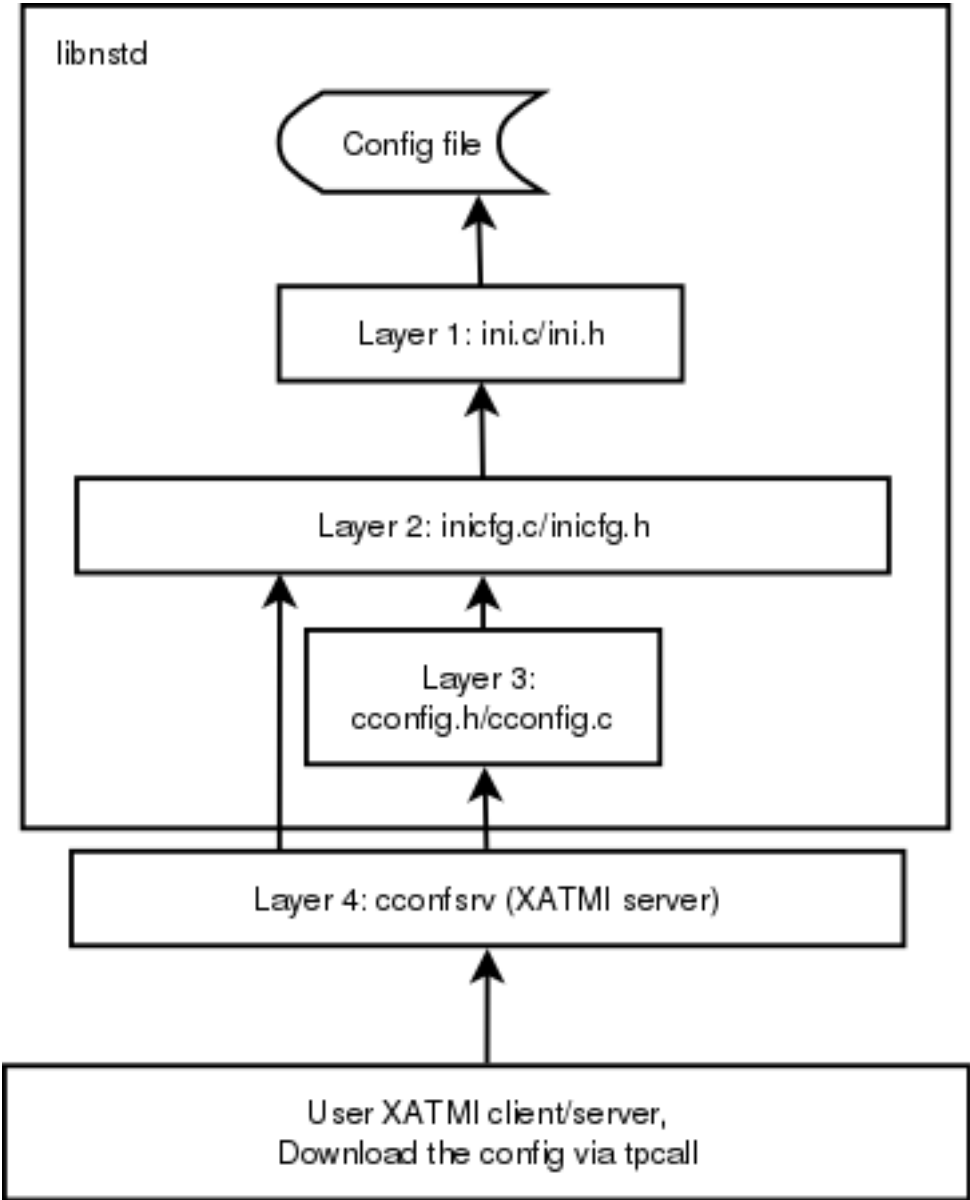
The idea behind this is that user can do the programming under Enduro/X in multiple languages (C/Go/Perl/Python/PHP/NodeJS) and these modules typically needs configuration. It would be waste of time if for each of the languages developer would need to think how to read the configuration from configuration 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/atmicht29.c` for sample code.

5.3 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.



Chapter 6

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)**.

6.1 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.

6.2 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)`**.

6.3 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

6.4 Brief of logging functions

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

6.4.1 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.

6.4.2 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.

6.5 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 */
```

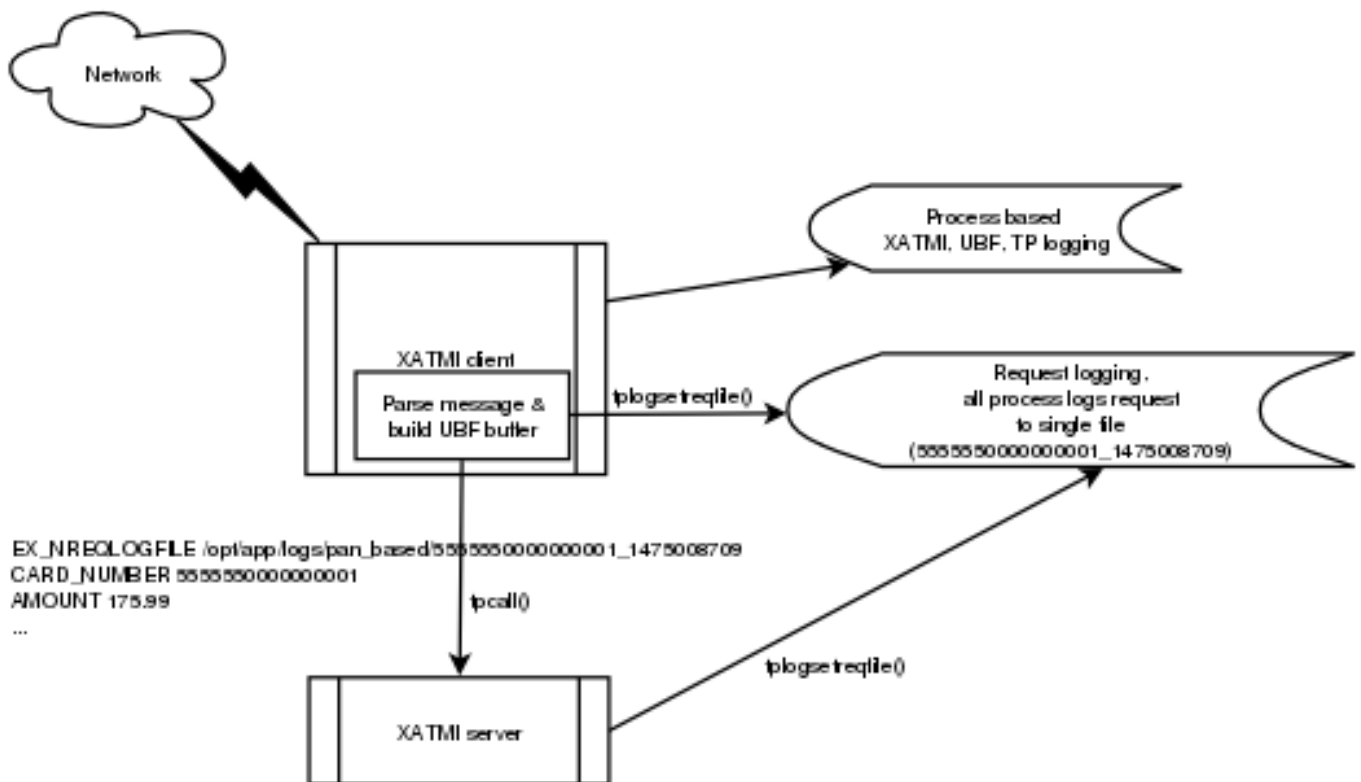
```

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

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

```

Let's assume that for our transaction logfile is set to: `/opt/app/logs/pan_based/5555550000000001_1475008709` then transaction could look like:



6.6 Understanding the format of log file

For example given code:

```

#include <nddebug.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:test1pc :11064:000:20160928:100225252:/tplog.c:0412:Hello from function logger
t:USER:5:test1pc :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>:<HOSTCR>:<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.
- *HOSTCR* - hostname crc32.
- *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.

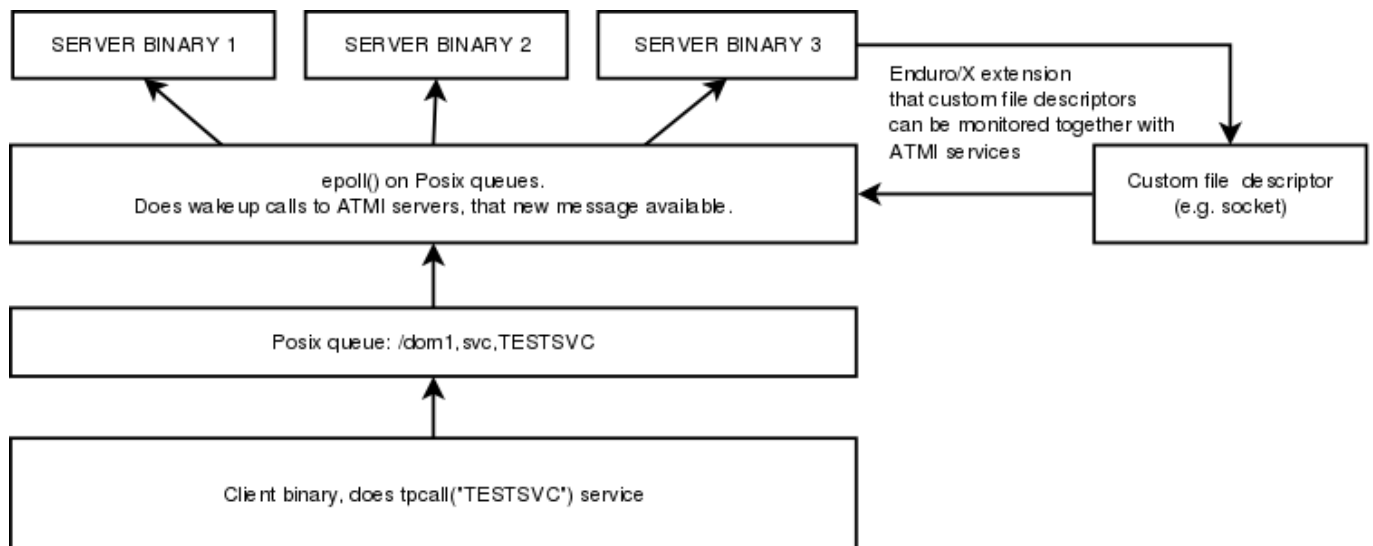
Chapter 7

Queuing mechanisms

This chapter describes different Enduro/X message transport mechanisms which are supported by Enduro/X. Enduro/X provides support for different transaction backends and there is no real difference for developer on which transport is used, except the performance and system resource administration aspects

7.1 Linux epoll and FreeBSD kqueue

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. FreeBSD operating system also supports polling and Posix queues. 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 and FreeBSD only.



7.2 System V mode

Starting from Enduro/X version 6, there are added sub-system for System V IPC queues. With this supports, good performance (about 70-80% from the epoll/kqueue) is reached for Unix platforms, i.e. Oracle Solaris and IBM AIX. The mechanisms as for epoll can give true single-queue multiple servers approach for perfect load balancing between local processes. By default each XATMI server process opens it's own job request queue, but this can be configured, so that "request address" is used, i.e. it defines one queue name on which all XATMI servers must advertise the same services.

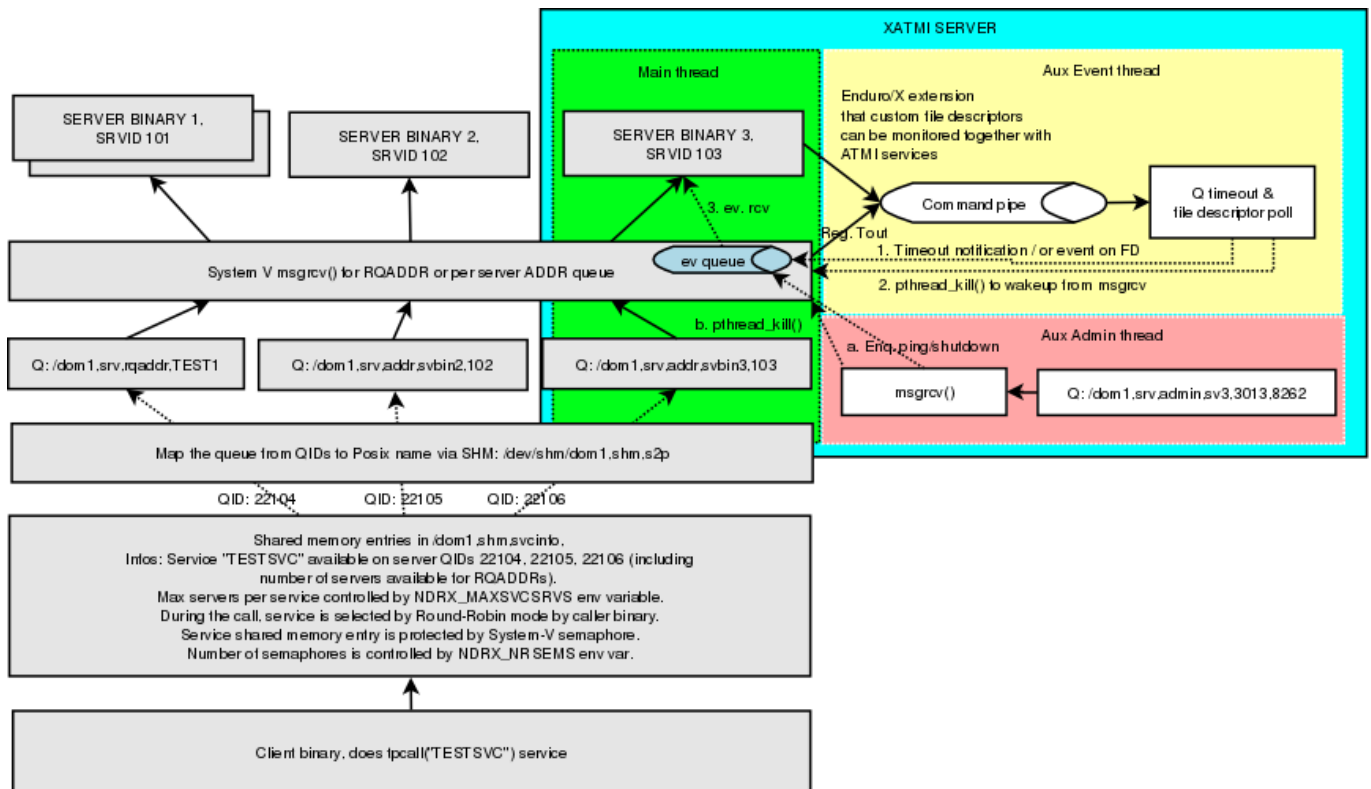
The implementation for System V is quite complicated, because these queues (see `msgrcv()`, `msgsnd()` system calls) does not offer any timeout control and secondly no form of polling on multiple queues are allowed. Thus these additional work semantics must be emulated.

Common approaches for **XATMI clients and servers** are following:

1. System V message queue IDs are mapped to string for queue names, which are used internally by Enduro/X. The mapping is done in two new shared memory segments whose size is controlled by **NDRX_MSGQUEUESMAX** environment variable, see `ex_env(5)`.
2. The timeouts are controlled by one new additional thread - event thread, to which via unnamed pipe particular queue descriptor pointer is submitted together with timeout. The thread then goes to sleep till the timeout. If the (main) thread which did queue operation is still in the queue call, it is interrupted by `pthread_kill()` system call. There are timestamp and sequence markers for each blocked System V IPC queue calls, so that timeout knows that main thread is in particular state, and have not reached the next call. If main thread has been woken up, then timeout is discarded once reached. Event thread performs timed poll on unnamed pipe, so that it is blocked till calculated next wake-up/timeout.
3. Special care shall be taken when process forking is required. As event thread is common for XATMI servers and clients, the fork by main thread will terminate those threads at un-determined point. Thus if new process wishes to continue to operate with XATMI session special approaches shall be done when forking Enduro/X processes. See sections in text regarding process forking.

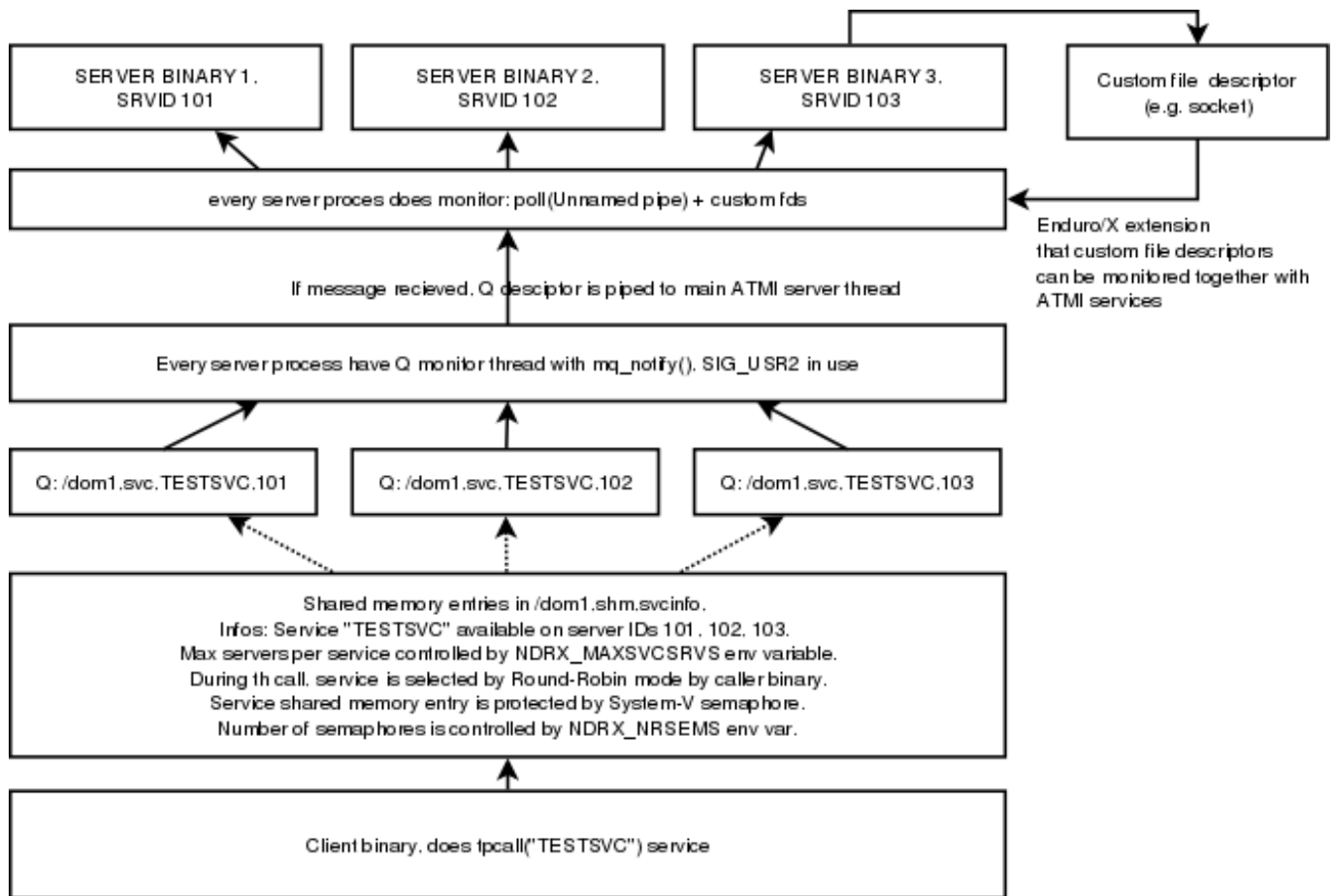
For **XATMI servers** the approach is further extended, so that:

1. There is one more additional thread listening for admin messages (i.e. pings and shutdowns). Once the message is received, it is placed in internal queue for the main server poller queue. If main thread is in blocked state of System V queue, it is waken up by `pthread_kill()`. If main thread was doing something else, then before doing next System V message receive, it checks the internal message queue and picks up the admin message from there.
2. As Enduro/X provides extensions for file descriptor polling, System V interfaces provides this API too. The user file extensions are put in the event thread's `poll()` structure. If event is noticed on user file descriptor, the event is sent to corresponding main thread and during that time, the file descriptor is removed from event thread, because otherwise it will signal again that there is something on the user FD (as main and event threads are async, and event thread might run faster than main thread). Thus user FD is enabled for polling only when main thread returns after the processing back to waiting for message.



7.3 Poll mode

Starting from Enduro/X version 3, there is support for other Posix compatible Operating Systems. Where possible Posix queues are used. If no Queue 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 separate 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 8

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 cdl;
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, "1", 0L, BFLD_STRING))
{
    ONDRX_LOG(&ctx1, log_error, "TESTERROR: OCBchg() failed %s",
              OBstrerror(&ctx1, OError(&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;
}
```

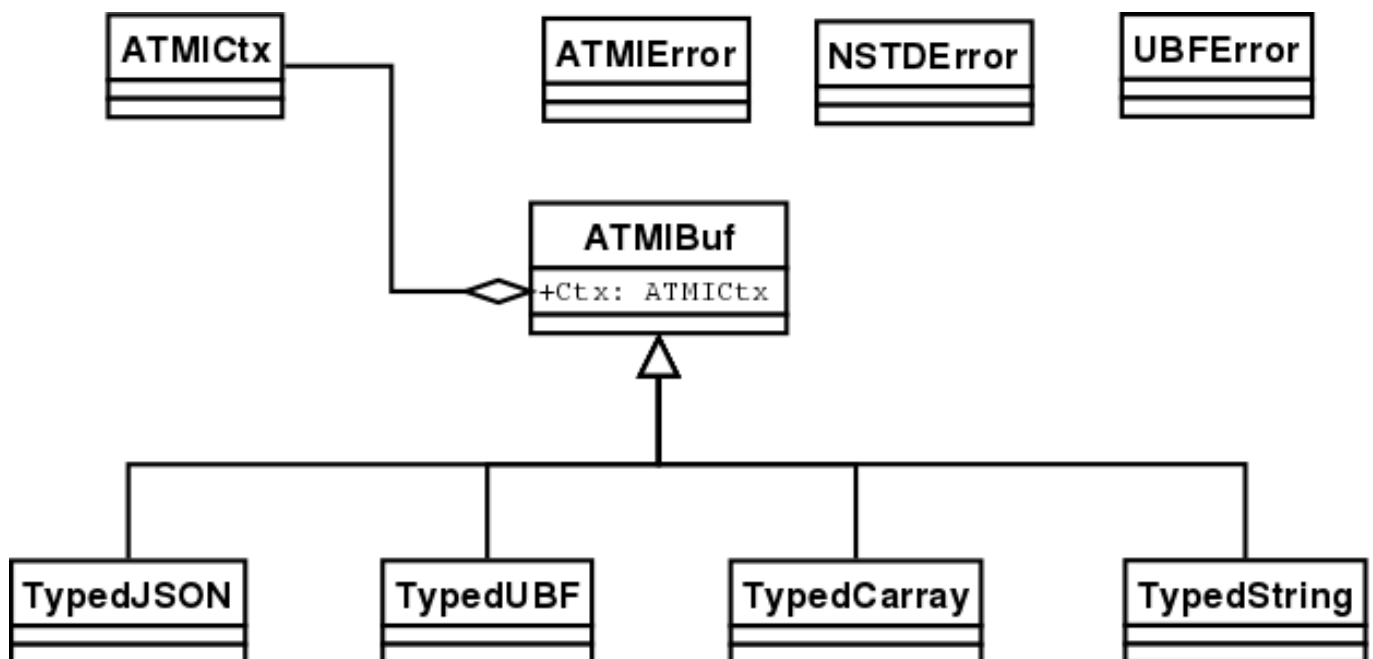
Build with:

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

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

8.1 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 9

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`.

9.1 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:

9.2 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).

9.2.1 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

```

9.2.2 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      bflldid      */
/*      -----      */
#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.

9.2.3 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.

9.2.4 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.

9.2.5 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 ndrconfig.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
```

$\frac{1}{\sqrt{\pi}} \left(\frac{x^2 - y^2}{2} + ixy \right) = \frac{1}{\sqrt{\pi}} z^2$

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 Apphome [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: ndrlexlev  :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: ucll       :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 Apphome [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 ndrlevel  :Logging: ATMI sub-system log level 5 - highest (debug), 0 - minimum ↵
    (off) [2]:
30: Edit ubflevel  :Logging: UBF sub-system log level 5 - highest (debug), 0 - minimum ( ↵
    off) [1]:
31: Edit tplevel   :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 ${NDRX_APPHOME}/log/testsv.log]:
39: Edit usv1_appopt :User server #1: appopt []:
50: Edit ucll      :Configure User client #1 [n]:
55: Edit addubf    :Additional UBFTAB files (comma seperated), can be empty [test.fd]:
56: Edit msgsizemax :Max IPC message size [56000]:
57: Edit msgmax     :Max IPC messages in queue [100]:
c: Cancel
w: Accept, write
Enter the choice [0-57, c, w]: w
ndrxconfig: [$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.fld** 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.fld .
```

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 $TESTHOME/log/tmsrv-rml.log -r -- -tl -l$TESTHOME/tmlogs/ ↵
    rml -- :
    process id=18855 ... Started.
exec tmsrv -k 0myWI5nu -i 41 -e $TESTHOME/log/tmsrv-rml.log -r -- -tl -l$TESTHOME/tmlogs/ ↵
    rml -- :
    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 -il -- :
    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.

9.2.6 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 10

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 *username, char *cltname, char *data, long len, long flags) ←
;
```

10.1 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:

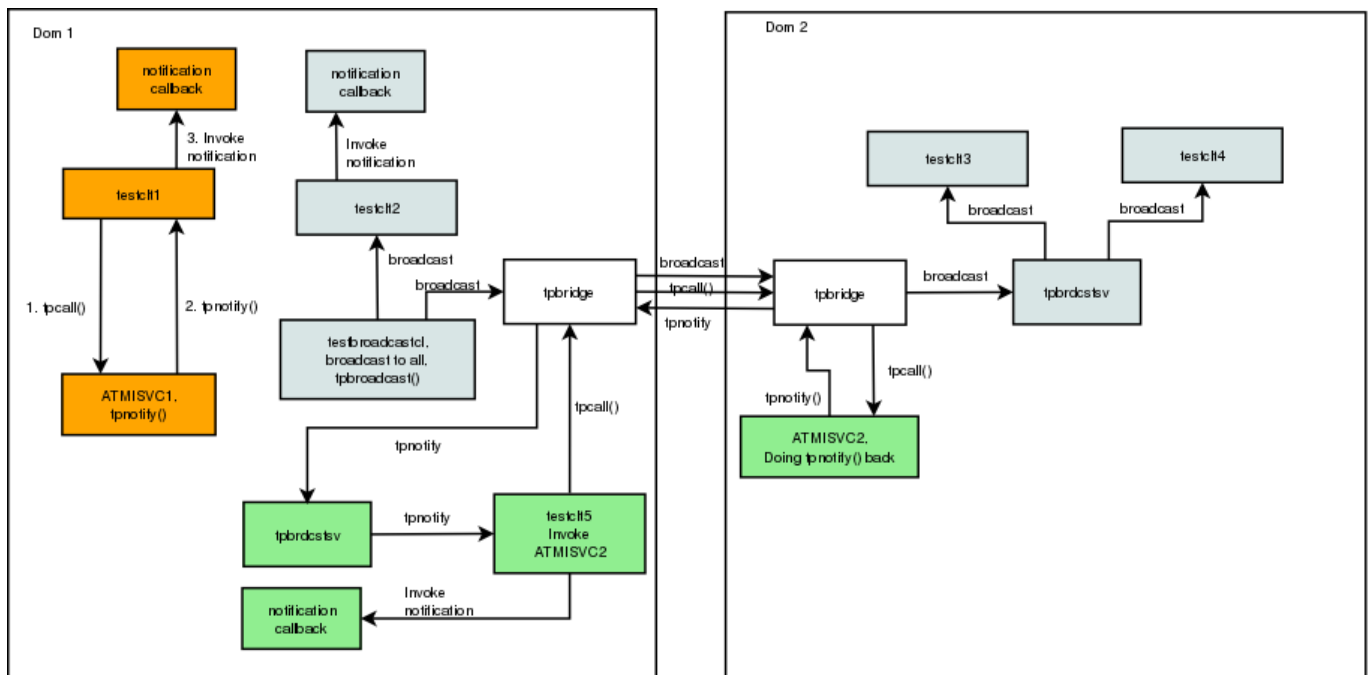
1. **tpalloc(3)**
2. **tpfree(3)**
3. **tpgetlev(3)**

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

10.2 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

10.3 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 11

Adding Enduro/X bindings

Currently Enduro/X have tier 1 bindings for the 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"! Thus 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 12

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 interface is provided via **expluginbase.h** header.

12.1 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 ndrplugin_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` (0x00000001) - plugin provides cryptography key function
2. `NDRX_PLUGIN_FUNC_TPLOGPRINTUBF_HOOK` (0x00000002) - hook for `tplogprintubf()` func

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.

12.2 NDRX_PLUGIN_FUNC_ENCKEY functions

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

```
int ndr_x_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).

12.3 NDRX_PLUGIN_FUNC_TPLOGPRINTUBF_HOOK functions

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

```
int ndr_x_plugin_tlogprintubf_hook(char **buffer, long datalen, void *dataptr1,
    int *do_write, FILE * outf, int fid);
```

Function is used to catch the moment when **tlogprintubf(3)** prints the UBF buffer in log file, line by line. For every line the function is called. The *buffer* receives the pointer to dynamically allocated string where current output data is formatted. The format of the line is following "<field_name>\t<any_data>\n". Total buffer length is passed in *datalen* field. *dataptr1* is reserved for future use. *do_write* by default is set to FALSE - 0, if set to TRUE 1, the line will be logged to the logging device. *outf* is current output stream. *fid* is UBF buffer field id currently being printed out.

The plugin function can capture the output, or it can replace the data in *buffer*, user is allowed to reallocate the *buffer*. Enduro/X does free() call on the *buffer* after the invocation.

See atmitest/test070_ubfhook/hookplugin.c for sample code.

Chapter 13

Starting Enduro/X XATMI server from other thread than main

For some scenarios it might be needed to create XATMI server to which main thread is busy with some other functionality. And only auxiliary thread may perform XATMI servicing actions. All this can be simply done with help of "libatmisrvinteg" and `ndrx_main_integra()` function. Thus following code fragment creates a simple server and provides command to build it under GNU/Linux.

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

#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#include <atmi.h>
#include <ubf.h>
#include <ndebug.h>
#include <unistd.h>

int M_argc;
char **M_argv;

/**
 * Service does not return anything...
 */
void HELLOSVC (TPSVCINFO *p_svc)
{
    tpreturn (TPSUCCESS, 0L, NULL, 0L, 0L);
}

/**
 * XATMI init callback
 */
int tpsvrinit(int argc, char **argv)
```

```
{
    NDRX_LOG(log_debug, "tpsvrinit called");

    if (EXSUCCEED!=tpadvertise("HELLOSVC", HELLOSVC))
    {
        NDRX_LOG(log_error, "Failed to initialize HELLOSVC!");
        return -1;
    }

    return 0;
}

/**
 * Do de-initialization
 */
void tpsvrdone(void)
{
    NDRX_LOG(log_debug, "tpsvrdone called");
}

/**
 * Run run_xatmi_server from thread
 */
void run_xatmi_server ( void *ptr )
{
    if (0!=ndrx_main_integra(M_argc, M_argv,
        /* set callbacks: */
        tpsvrinit, tpsvrdone,
        0L))
    {
        NDRX_LOG(log_error, "Failed to run Enduro/X main: %s",
            tpstrerror(tperrno));
        exit(1);
    }
}

/**
 * Standard main entry...
 */
int main(int argc, char** argv)
{
    pthread_t thread1;
    pthread_attr_t pthread_custom_attr;

    M_argc = argc;
    M_argv = argv;

    pthread_attr_init(&pthread_custom_attr);

    /* Configure stack, using Enduro/X internal method...
     * but you can configure it by your self.
     */
    pthread_attr_setstacksize(&pthread_custom_attr,
        ndrx_platf_stack_get_size());

    pthread_create (&thread1, &pthread_custom_attr, (void *) &run_xatmi_server, NULL);

    pthread_join(thread1, NULL);
}
```



```

    return 0;
}

```

To compile the code, you may just use C compiler:

```
$ cc -o samplesv sample.c -latmisrvinteg -latmi -lubf -lnstd -lpthread -lrt -ldl -lm
```

To boot the server, you may add it to `ndrxconfig.xml` and boot it up:

```

...
    <servers>
...
        <server name="samplesv">
            <srvid>1600</srvid>
            <min>1</min>
            <max>1</max>
            <sysopt>-e /tmp/SAMPLE_1 -r</sysopt>
        </server>
...
    </servers>
...

```

Finally give it a test:

```

NDRX> start -y
...
exec samplesv -k 0myWI5nu -i 1600 -e /tmp/SAMPLE_1 -r -- :
    process id=9650 ... Started.
...

NDRX> psc
Nd Service Name Routine Name Prog Name SRVID #SUCC #FAIL MAX LAST STAT
--
...
1 HELLOSVC HELLOSVC samplesv 1600 0 0 0ms 0ms AVAIL
...
NDRX>

```

NOTE: the process is started with standard command line with args like `-k/-i/-e` and `--`. If your existing software also uses CLI arguments, then it must be modified so that it does not crash with unknown keys. Also Enduro/X XATMI server will not tolerate any other third party keys. Those other keys user might add it "appopts" section, followed by `--`. See `ndrxconfig.xml(5)` for more details.

13.1 Enduro/X Process naming strategies

The process naming strategies are complex ones, due to fact that server processes can be booted for shell-scripts and Enduro/X sees the upper level script name and not the actual binary which performs the XATMI work. The process name participates in following functional areas:

1. For clients, is opening of client's reply queue
2. For servers, it is used for admin queues and reply queues.

The **ndrxd** daemon performs sanity checks, and it is doing cross-validation, that server queue with process name X and PID Y exists in system. While for queue cross validation, all is ok, as X and Y exists, the problem is with cases when **ndrxd** performs server PINGs. To build admin queue name to send ping to, **ndrxd** uses the binary name it knows (either server name or real name extracted from command line). As command line can contain shell script, **ndrxd** will not be able to send PINGs to server process, as queue will be different, and then **ndrxd** will kill the process as not pingable.

13.1.1 Strategy 1

To cope with above cases, Enduro/X uses following solution:

1. If NDRX_SVPROCNAME is exported, then server process opens queue with this name.
2. In normal case NDRX_SVPROCNAME matches the server binary name, thus no changes for existing system logic.
3. In case of Java env, the *argv[0]* will be set to this environment variable or it will use just keyword "java" if variable is not available.

To check that process exists, for servers this could be done in this way:

1. Check the PID existence (extracted from Q name)
2. Check the Process name existence (extracted from Q), does it match the PID (exec in the same approach of first test)
3. If does not exists, lookup the Process Model (PM). If the extracted name matches process name in PM, and the name from CLI matches the real name, the process exists.
4. If above does not work out, lookup the environment variables of the process, check the existence of NDRX_SVPROCNAME variable. If value matches the name extracted from queue, then process is alive.
5. Otherwise process is dead.

13.1.2 Strategy 2

The server process reports the final process name to the **ndrxd** while it reports it's status and advertised services. At this point **ndrxd** may start to send pings to server. Regarding of pinging non reported servers, this is up to **ndrxd** current algorithms of when to ping.

Initially we take server name or exe name from command line as one to which admin messages shall be sent (prior receiving the name from the process).

The Strategy 2 is simpler to implement. The Strategy 1 shows the server queues as virtual process names. That might be simpler for admins to understand to whom the queue belongs to.

13.2 Booting processes as XATMI servers with out CLOPT

There could be server processes like Tomcat or JBoss App servers which we might want to boot as XATMI servers. For these command line options cannot be passed in. Thus at `ps -ef` the output will not show Enduro/X specific flags like `-k` (unique app key), `-i` (server instance id), `-e` (error log), etc.

One approach would be to pass these command line options in NDRX_SVCLOPT variable. But again we will have an issue with some functional areas - like "xadmin down" command.

This could be solved by doing peek into other process environment. If we find the unique string there - then server process is subject for killing.

Chapter 14

Process forking

There are special moments to take care of when Enduro/X process either XATMI client or XATMI server want's to perform forking (i.e. duplicate process).

Enduro/X offers following APIs to support better forking flow:

1. **ndrx_fork(3)** which is wrapper for **fork()** call which includes any resource deallocations/allocations necessary.
2. Another way if stock **fork()** needs to be used, then Enduro/X supports following APIs: **ndrx_atfork_prepare()** - parent calls before fork, **ndrx_atfork_parent()** - parent calls after fork, **ndrx_atfork_child()** - child calls after fork.

There are three kind of issues that needs to be solved when forking.

1. In case of XATMI server process, when child process is forked, any server queues must be closed to avoid memory leaks. This can be solved by either using **ndrx_fork(3)** or by calling **ndrx_atfork_child()**.
2. If client process (or server process with threaded clients) is forked, then clients must be freed by **tpterm(3)** manually by developer.
3. In case if System V queues are used, auxiliary threads must be terminated, resources released and auxiliary threads must be resumed for parent process. This can be solved either by using **ndrx_fork(3)**, or **ndrx_atfork_prepare()/ndrx_atfork_parent()/ndrx_atfork_child()**.

Functions **ndrx_atfork_prepare()/ndrx_atfork_parent()/ndrx_atfork_child()** can be registered with **pthread_fork()** so that library fork calls are supported.

WARNING! When operating in System V mode and user is performing **ndrx_fork(3)**, any other user threads shall not perform XATMI IPCs (tpcalls, tprecv, etc..) in any other concurrent user thread, otherwise unexpected process corruption might happen. Thus corresponding synchronization shall be done at user code.

Chapter 15

Source code management

This section lists notes for Git usage for Enduro/X development. Usually we modify the sample configuration in directory **sampleconfig** in order to get the test system working. But these changes should not be committed, as mostly they are local means and local config. Thus to avoid the changed files from auto commit when using **git commit -a**, files can be marked as "unchanged" by following git command:

```
$ git update-index --assume-unchanged sampleconfig/debug.conf
$ git update-index --assume-unchanged sampleconfig/ndrxconfig.xml
$ git update-index --assume-unchanged sampleconfig/setndrx
```

Chapter 16

Process debugging

This section lists some notes about techniques for debugging memory issues for the binaries.

16.1 Tracking down memory usage with Valgrind for XATMI servers

Valgrind utility can be used with XATMI servers to get some insight in memory usage and leaks. Here is the sample XATMI server definition which uses Valgrind wrapper for server boot.

```
<server name="myserver">
  <srvid>100</srvid>
  <min>1</min>
  <max>1</max>
  <sysopt>-e /tmp/MYSERVER -r</sysopt>
  <appopt>-c10</appopt>
  <cmdline>valgrind --leak-check=yes --log-file=/tmp/valg.log ${NDRX_SVPROCNAME} ${ ←
    NDRX_SVCLOPT}</cmdline>
</server>
```

Chapter 17

Two Phase Commit Processing notes

This section describes various aspects of distributed transaction processing.

17.1 Normal Enduro/X Operations of Two Phase Commit

The processing sequence of distributed transaction from XATMI point of view is following:

- For each Resource Manager (RM, e.g. database), there is configured **tmsrv(8)** process.
- XATMI process connects to RM with **tpopen(3)** function call
- XATMI process starts transaction with **tpbegin(3)** function call. When this request is issued, internally this calls **tmsrv(8)** for requesting the new transaction XID. At this point tmsrv start accounting of the transaction. And XID is returned to the caller process.
- XATMI process calls starts transaction by call of `xa_start()`. Previous versions of Enduro/X used **TMJOIN** flag, because tmsrv did already start the transaction in database. With 6.1 release, **TMJOIN** is not used, because tmsrv only generates XID, and does not perform any work with database.
- In case of process doing `tpsuspend(3)` call, the `xa_end()` function with **TMSUCCESS** is called. The suspend data can be used by other processes on the same architecture machines to resume the transaction.

17.2 Non XA Switch ready database

There are several databases which provide incomplete support for X/Open Groups XA Standard implementation. For example Mysql/MariaDB or Postgresql. In case of these databases, process cannot join existing transaction with `xa_start()`. Thus here every time process attempts to join TX, new transaction id is requested from master **tmsrv**. The **tmsrv** logs this id in journal file, because even if transaction is not prepared, for Postgresql prepare is only way to report the XID to database.

17.3 TMSRV Transaction log format

Currently there are two versions of the log, v1 and v1.1. V1.1 version is extended for Multi TID per RM support. Transaction log file consists of several record types. It logs general info about transaction, information about current state and each Resource Managers (RM) status. The file format is simple. It is text file with Unix newlines. Each record is written on each text line. The field separator in record is colon ":". Each record starts with Unix UTC timestamp in milliseconds. Log handling is done in **tmsrv/log.c** source code. Following record types ("log commands") are used:

- **I** - General Information about transaction
-

- **S** - Identify stage of transaction (is it active, committing, abort, etc.)
- **R** - Resource manager status information (is it active, prepared, committed, etc.)

17.3.1 General info about the transaction

This gives basically some initial infos about transaction. Which tmsrv created it, what is the timeout setting.

Nr	Field name	Type	Mand/Opt	Version	Description
1	tstamp	N13	Mand	1	UTC Time stamp when record added
2	cmdid	Const "I"	Mand	1	Record type identifier
3	tmmid	N..5	Mand	1	Resource manager ID which have started TX
4	tmnodeid	N..5	Mand	1	Cluster Node id which started transaction
5	tmsrvid	N..5	Mand	1	Transaction manager server id (srvid from ndrxcnfig.xml)
6	txtout	N..19	Mand	1	Transaction timeout set by tpbegin(3)

17.3.2 Transaction stage identification

Nr	Field name	Type	Mand/Opt	Version	Description
1	tstamp	N13	Mand	1	UTC Time stamp when record added
2	cmdid	Const "S"	Mand	1	Record type identifier

Nr	Field name	Type	Mand/Opt	Version	Description
3	txstage	N..2	Mand	1	0 - transaction does no exists, 5 - transaction is active in processing, 20 - transaction is aborting, 25 - transaction aborted with hazard, 30 - heuristically aborted, 35 - aborted ok, 40 - preparing, 55 - committed (hazard), 65 - heuristically committed, 70 - committed ok

17.3.3 Resource manager status information

Each resource manager (database, etc) has it's own status distributed transaction, either it is joined to transaction, prepared, committed, error. If resource manager supports **TMJOIN**, then only one record is made for resource manager. If resource manager does not support **TMJOIN**, then each time thread starts to process, new transaction id is requested. Thus it is logged separately to log file. This functionality is starting with record version 2.

Nr	Field name	Type	Mand/Opt	Version	Description
1	tstamp	N13	Mand	1	UTC Time stamp when record added
2	cmdid	Const "R"	Mand	1	Record type identifier - resource manager status
3	rmid	N..5	Mand	1	Particular ID of resource manager for which status is being tracked

Nr	Field name	Type	Mand/Opt	Version	Description
4	rmstatus	A1	Mand	1	0x0 - NULL, n - Non transaction, i - Idle state (not used), j - RM is in joined state (active), p - RM is in prepared state, a - RM is in abort state, b - Aborted (heuristically), d - Aborted (hazard), c - Committed, r - Committed (was read only), h - Committed (Heuristically), z - Hazard (committed or aborted)
5	rmerrorcode	N..3	Mand	1	TP Error code when RM entered in this status. 0 - means no error occurred.
6	rmreason	NS..4	Mand	1	XA Error code for caused RM status.
7	btid	N..19	Opt	2	Transaction id, per resource manager. If value is not set, then it defaults to 0

numbered!:

Chapter 18

Additional documentation

18.1 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 19

Glossary

This section lists

ATMI

Application Transaction Monitor Interface

UBF

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