CAOS ANET

Dmitri Kondratiev

dkondr@bigfoot.com

Table of Contents

This document describes CAOS ANET - CAOS Active Network platform architecture. Document is far from being complete and at this time mostly provides high level overview of the system. Corrections and additions are to be provided by ANT and SC teams in future versions of the document. This document also suggests the logical incremental order of new CAOS ANET system implementation.

CAOS and ANET goals

CAOS - Cambira Operating System is GSN OS. CAOS includes both traditional router OS components such as routing and packet management subsystems as well as active network platform - ANET. The main goals of CAOS ANET are:

- Provide high performance distributed operating environment for 'intelligent' router capable effectively solve both traditional routing tasks as well as perform application-based routing in active network (AN).
- Provide scalable and flexible routing OS where both non-active and active components could coexist. It should be easy to add/remove new functionality to the system without reboot. It should be possible to migrate non-active components to ANET as needed.

Design principles

The main principles of CAOS ANET design are:

- CAOS ANET is a distributed OS tightly integrated with QNX Neutrino microkernel. This allows exploit to the most QNX real-rime support and QNX networking.
- Provide close to wire-speed packet processing based on data 'zero-copy' mechanism deployed throughout all system components.
- Multi-process architecture. Every active component, both Active Applications and traditional router components are realised as separate distinct CAOS/QNX process.
- Provide optimised for speed communication between processes.
- Provide secure operating environment with a flexible framework of system resource accounting and support for different resource usage policies.
- Persistent System State. Any CAOS process should be able to serialise its state to non-volatile storage (ROM / Flash) and deserialize from it later at boot up. As a result, at any given time, it should be possible to restore state of the complete system.
- Recovery Oriented System. Capable to recover from most failures of one or more CAOS processes as well as QNX IONET.
- Highly configurable in run-time ANET component architecture. Active Application (AA) components (elements) can be combined in run-time in AAs with desired properties using channels. This provides for highly customisable behaviour of 'intelligent' router that can be achieved without halting and rebooting the system.
- In run-time provide support to dynamically load program code from other CAOS ANET nodes.

- Provide robust development environment with Channel Element Language (CEL) generating highly-optimised AA code.
- Provide system-wide support for flexible trace-log and debugging facilities.
- Design should allow for future incremental additions of functionality to the system, such as support for security model based on code digital signatures and security policies.

Building blocks of the system

The following components are used to build all CAOS ANET modules and subsystems, and are in fact 'building blocks' of the system:

- QNX Neutrino micro-kernel.
- QNET networking.
- CAOS Process (CP).
- CAOS Process Manager (CPM).
- Shared Memory Manager (SMM)
- Smart Messages Framework (SMF).

CAOS process and Caos Program Manager

CAOS process (CP) is a main component of the system. Both AA and non-active programs exist in the system as CAOS processes. CAOS process is an extension of QNX POSIX process. CAOS ANET introduce this new type of process to be able to:

- Register and account all running CAOS processes in the system.
- Detect CAOS process failures and provide restart.
- Constrain process resource usage, such as memory, stack, threads according to different resource usage policies.
- Associate arbitrary meta data with every process that will allow to identify and deal with CAOS processes in more intelligent way.

CAOS Program Manager (CPM) extends QNX process manager (procnto) and both extends existing and provides additional functions to manipulate CAOS processes (CP) to:

- Control maximum stack size and shared memory allocation at process start.
- Control thread allocation.
- Control scheduling policy and scheduling parameters (priority).
- Provide 'software watch dog' to detect process failures and signal other system components (CHAM - Caos High Availability Manager) about these events.
- Associate meta data with the process.
- Registration and discovery of the process according to associated meta data.
- Control node to create process on.
-

Shared Memory Manager

Shared Memory Manager (SMM) provides system-wide shared memory management. Shared memory is used by all CAOS process for efficient "zero-copy" data transfer both in context of one process as well as across processes. In fact all transfers involving data buffers equal or bigger then 128 K (on Intel architecture) are realised with passing pointers to these buffers both inside single process and across processes.

Interface to SMM is realised with SIM library that provides access to SMM functions including:

- Allocate collections of shared memory objects / slots of fixed size and map them in process space. This collection follows format similar to e-mail message box, or SMBOX in short. When SMBox is created it is mapped in requesting process space.
- Find existing SMBox. This returns SMBox ID already mapped by some process.
- Map SMBox in this process space. Returns pointer to SMBox buffer.
- Deallocate SMBox.
- Access slots in SMBox. This should return slot address in this process space.
- Get number of slots in SMBox.
- Get slot size in octets.

More details on SIM shared memory library to be provided by SC Team.

Smart Messages

Smart Messages (SM) - messaging framework built with standard QNX Neutrino messages with the goal to provide optimised for efficiency data transfer both for local and distributed communications. Other goals that SM framework address are attribute-based messaging, blackboard and asynchronous communication, message 'store and forward' mechanisms. SM uses powerful QNX messaging to argument traditional shared memory and other IPC and provide user with efficient and flexible synchronisation and data communication mechanism to be used in uniform way no matter where communicating end-points exist: in separate threads of a single process, in different processes running on the same node, or in different threads running on different nodes on the net.

Smart Messages framework address the following goals:

- Provide universal naming space and addressing scheme both for local and distributed message senders and receivers or message end-points.
- Provide distributed registration / discovery mechanism based on message end-points attributes.
- Provide message 'store and forward' mechanism when this is requested by end-points.
- Provide efficient transfer of message data optimised for speed with smart usage of local and shared memory.
- Provide simple and robust synchronisation mechanism for communicating end-points.

Basic mechanism

Smart Messages are built with native QNX message passing IPC provided in Neutrino kernel. Message passing is fundamental Neutrino IPC with all other forms of IPC (pipes, FIFOs, etc.) built over it.

Communicating end-points

In Neutrino addressable message end-point, called 'channel', is defined with a three element tuple (NodeID, ProcessID, ChannelD). This allows the following types of communication between senders and receivers of messages:

- Sender and receiver running in different threads of the same process.
- Sender and receiver running in different processes on the same node.
- Sender and receiver running in different processes on different nodes.

To start message exchange receiver must create one or more channels that sender will use to send messages. Neutrino natively support 'fan-in' messaging mode, when receiver may get messages from different senders on the same channel simultaneously.

Receiver registration and sender connections.

When receiver creates a channel, corresponding message end-point (NodeID, ProcessID, ChannelD) gets registered with local QNX Program Manager (PM). To start sending messages any sender needs to connect (ConnectAttach()) to the desired end-point. Again, many senders may connect and send messages to the same end-point.

Sending and receiving messages

To send messages senders use MsgSend() family of Neutrino messaging functions while receivers use MsgReceive() function set. In general case of sending messages inside a single process Neutrino kernel copies a message directly from the address space of one thread to another without intermediate buffering. Thus, for local case, the message-delivery performance approaches the memory bandwidth of the underlying hardware.

Neutrino also provides functions for multi-part transfers. With these functions both the senders and receivers can specify a vector table (IOV) describing message memory fragments that both senders and receivers can use. Also provided is a special case of multi-part message that is a simple single-part message with a pointer to a single message buffer. All these message functions can be used to pass pointers to data buffers between senders and receivers running in a single process without coping data itself.

Synchronisation

Neutrino provides simple, clear and powerful mechanism to synchronise message senders and message receivers based on MsgSend(), MsgReceive(), and MsgReply() primitives.

- 1. Receiver gets blocked waiting for message with MsgReceive().
- 2. Sender sends message with MsgSend() and gets blocked waiting for reply from receiver.
- 3. Receiver gets message and gets unblocked returning from MsgReceive(). Sender remains blocked.
- 4. Receiver now can implement any appropriate strategy required by message processing protocol that was agreed on with sender. Also receiver has full control when exactly to unblock the sender. In some cases sender may stay blocked until receiver fully processes the message. In other cases it may be needed to unblock receiver as soon as possible. No matter what strategy receiver needs at this step

Neutrino guaranties that sender will stay blocked and thus will not be able to alter the message until receiver fully process the message and explicitly unblocks receiver.

- 5. Receiver is done with message processing and unblocks sender with MsgReply() call.

Extending basic mechanism with Smart Messages

Smart Messages (SM) extend basic Neutrino messaging in many powerful ways while preserving Neutrino messaging flexibility through SM control interface.

SM end-points addressing

In addition to standard Neutrino addressing of message end-points based on three-element tuple (NodeID, ProcessID, ChannelD), SM provides 'wild card' addressing based on attributes associated with every end-point (Neutrino channel).

SM aware processes can register channel end-points together with associated set of attribute value pairs (AVP). As a result :

- SM Senders can query receivers according to the attributes they publish not only by their names.
- Sender can connect and send messages not only to some other very well known process end-point but to 'any' end-point that matches attributes requested by the sender during connection phase.
- Smooth visioning. New receivers with the same attributes can replace old versions of the same receivers dynamically in running system.

Sending and receiving messages with SM. Message 'store and forward'

In addition to standard Neutrino messaging SM provides message 'store and forward' facilities. This can be used for senders that need to post messages when receivers are not ready yet to process these messages without blocking senders. Very well-known concept used in e-mail systems.

SM framework provides all necessary control both for servers and receivers to enable and parametrise their 'store and forward' capabilities as needed.

Both attribute-based addressing and 'store and forward' facilities allow for processes to create and use customised Blackboard communication models with different parameters of these to be fine-tunned both by senders and receivers.

Optimised for efficiency data transfer with SM

SM provide flexible and powerful mechanism for efficient, optimised data transfer. SM framework may be used by its clients (senders and receivers) for automatic and transparent handling of message data buffers. In this 'optimised' mode SM framework itself detects the collocation of sending and receiving end-points and choses the fastest way to transfer the data taking the burden of this work from the clients.

Another, 'explicit' mode allows sender to explicitly request SM framework to pass a message either as a pointer to message data or as an actual copy of the data. In case 'pointer mode' does not make sense as in case with remote receiver running on a separate node, SM framework will return appropriate error code to the sender.

To optimise data transfer SM framework :

- Provides clients with means to create custom message protocols specialised for the particular application needs. Specialisation here allows to minimise protocol overhead that general-purpose protocols involve and make the protocol 'just right' in other words as simple as possible for any given set of application senders and receivers.
- Provides minimal base protocol that allows receiving part of SM framework to determine the type of message buffers sent: fragmented pointer-based buffer (IOV), data copy or shared memory object.
- Transparently makes optimisation decisions for data transfers between processes running on the same node. In case any unfragmented message buffer is equal to 2/3 of the MMU page size, framework maps this buffer to shared memory object on sender size and remaps it back to receiver process memory. Otherwise message buffer is copied between processes.

Note: The size of the data buffer when this decision is made as well as many other SM parameters can be configured by application.

SM Synchronisation.

Message 'store and forward' mechanism adds new synchronisation model to basic Neutrino synchronisation with MsgSend(), MsgReceive(), and MsgReply() primitives also 'inherited' by SM.

Senders and receivers may explicitly request SM framework to use 'asynchronous' message passing mode. In this mode senders post messages to SM framework to be retrieved any time later by receivers. Senders can also specify message expiration interval during which messages will be waiting to be consumed. Thus both senders and receivers can now transfer messages asynchronously.

SM Summary

Smart Messages extend Neutrino messages in many powerful ways that make possible attribute-based messaging, blackboard and asynchronous communication, message 'store and forward'. Besides SM allows to optimise message communication for speed and efficiency thus in most cases eliminating the need for other IPC including shared memory which SM utilises in efficient way transparently for its clients.

System architecture

CAOS ANET is a laired architecture that defines the following subsystems:

- Neutrino micro-kernel and system processes (procnto, io-net, etc.)
- Routing subsystem that includes Packet Manager, Classifier, Scheduler and Route Lookup. *To be verified by SC team.*
- Cambira Virtual Machine (CVM) - provides run-time environment for Active Applications
- Active Network Services (ANS) - active applications that provide services to other components of the system. These include AAs that implement various protocols, such as ARMTP, TFTP, RTP etc.
- AA that provide configuration/customisation of AN services in run-time.

- Resource Manager (RM) - registers and controls access to CAOS ANET resources (processes, threads, memory and bandwidth) allocated to every CAOS Process. Also provides system-wide Resource Browser.
- Dynamic Code Loader (DCL) - process that allows other CAOS processes to load and start code from local or remote file storage. In future implementation DCL and CPM will support 'secure code' model. Secure code model will allow loading and starting digitally signed code from third-party developers.
- CAOS High Availability Manager (CHAM) - monitors Neutrino and CAOS processes and performs multistage recovery whenever any processes fails or no longer respond. CHAM works in close cooperation with CAOS Program Manger (CPM)
- CAOS System Manager (CSM) - process that allows human operator to define different policies for RM and CHAM and also configure/control ANS.

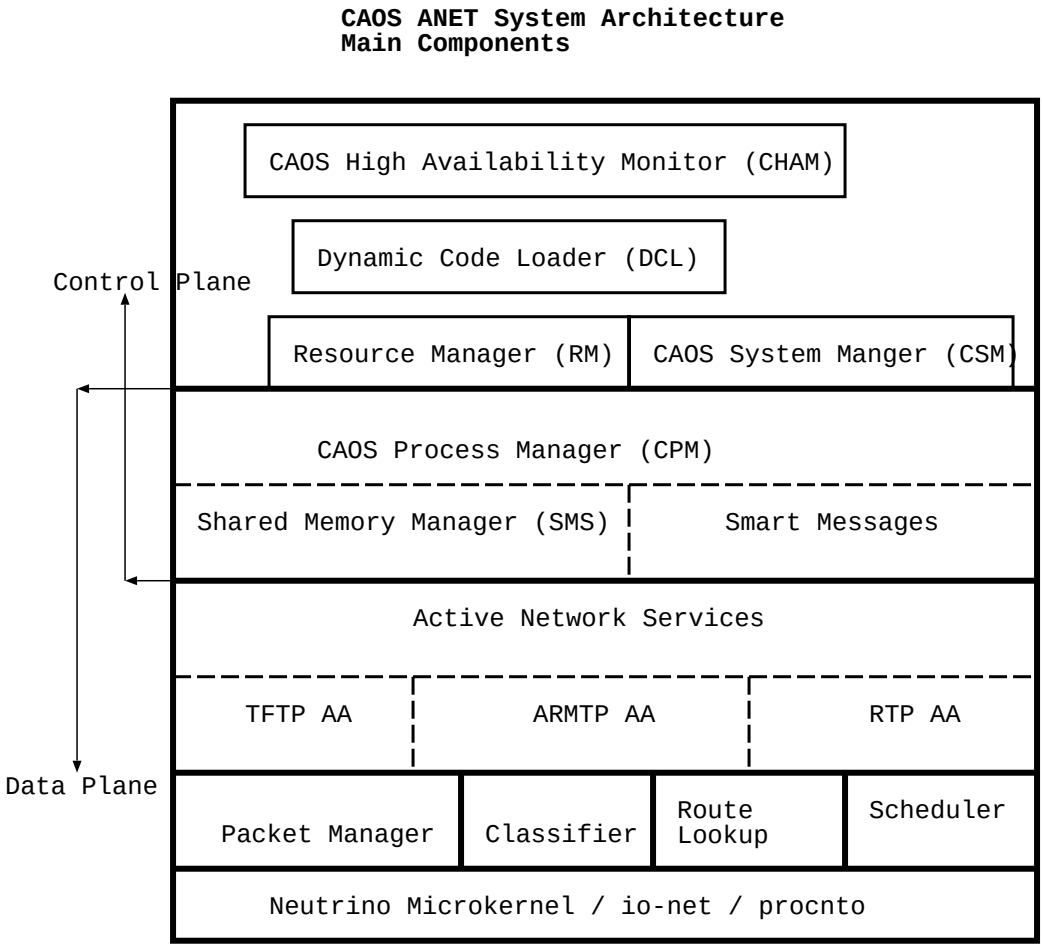

Figure 1. CAOS ANET System Architecture. Main components.

ANET development tools and environment

ANET development tools are programming language and debugging tools necessary for rapid development of AA in CVM environment:

- Channel Element Programming Language (CEL) - high level CVM Channel programming language and translator. CEL provides constrained programming environment that allows effectively program channel elements and eliminates memory allocation and bad pointers errors. CELT - CEL translator generates highly optimised C / C++ code ready for optimised compilation with GCC.
- CDB - channel debugger allows remotely debug AA in run-time.
- DCL - dynamic code loader allows to load CVM channel libraries in run-time and perform dynamic linking of new CVM channels with existing running CVM AA channels.

Further design and implementation order.

To move forward with new CAOS ANET system design and implementation incremental approach is suggested whith the following order of development (where items on one line are developed in parallel) :

- SMM (SIM), SM, CAOS Process Manager (CPM), CP
- RM, CEL, CDB, DCL
- CSM, CHAM