Janos Component Manual

Flux Research Group
Department of Computer Science, University of Utah

janos@flux.cs.utah.edu

May 4, 2000

This document applies to the Janos Java Libraries v1.0 and later snapshots containing minor updates.

The released Janos code is currently all in the `java/` subdirectory. There are currently two major components to that code:

- Janos Java NodeOS: Java bindings of the NodeOS abstractions. Reside in the `nodeos/` directory.
- ANTSR: **ANTS** with **R**esource Management: our port of ANTS to the NodeOS abstractions, as well as internal changes to accommodate precise resource control. Resides in the `antsr/` directory. All of the ANTSR applications and utilities are in different subdirectories of `antsr/`.

The `route/` directory contains a simple implementation of the NodeOS-level routing table. Its a simple table that maps addresses to route objects. The `io/` directory contains the logging interfaces provided for EEs built on our NodeOS. The `util/` directory contains utility code for EEs running on the Janos NodeOS interface.

1 Janos Nodelib

The current NodeOS Specification[1] specifies the following standard interfaces and abstractions. The "+" and "-" marks indicate the current degree of support in our Java binding.

+ Flow: the unit of accounting and control
+ ThreadPool: for controlling thread concurrency, interface to threads
+ Channels: sources and sinks for network traffic.

+/- Packets: for manipulating network traffic elements.

+/- FileSystem: interfaces to persistent storage

- MemoryPool: groups memory for inter-flow sharing.
- Events: for non-packet related timers and scheduling.

Of these interfaces, the Janos Java NodeOS currently provides Java bindings for the Flow, ThreadPool, and Channels. A Buffer abstraction that works at a higher level than the Packet abstraction is also provided. The Java bindings in this interface are meant to simultaneously take advantage of Java's direct

[1] *NodeOS Interface Specification*, Active Network Node OS Working Group, Draft of January 24, 2000. Available at `http://www.cs.princeton.edu/nsg/papers/nodeos.ps`

support for objects (e.g., utilizing constructors) and hide NodeOS details that need not be exposed at this level (e.g., the memory page mapping interfaces). To these ends, on some details the mapping deviates from the C bindings defined in the NodeOS specification. The goal of this interface is that it be trivial to implement *on top of* the specified NodeOS C API, while only exposing the details that an EE needs. (The current implementation is entirely in Java, using the standard JDK 1.1 facilities.)

Several of the names chosen for Janos NodeOS Interface functions are different than their C API NodeOS counterparts. Some of these will be changed to reflect the NodeOS name; others are more in line with Java conventions and will remain. Details are provided later.

The javadoc-style documentation of the actual Java interfaces is available at `http://www.cs.utah.edu/flux/janos/apidoc/`.

1.1 Comparison to NodeOS C bindings

This section provides a function-to-function comparison of the C language binding of the NodeOS API with the equivalent Janos NodeOS interface function. This section will make sense only if you are *very* familiar with the NodeOS specification, or if you have a copy of it next to you.

Each of the Janos classes listed below is in the `edu.utah.janos.nodeos` package, e.g., the fully qualified name of the method `Flow.newFlow(...)` is `edu.utah.janos.nodeos.Flow.newFlow(...)`.

1.1.1 Flows

The Janos Flow abstraction is designed to directly map a NodeOS flow's capabilities into Java. There are no semantic changes introduced other than the `FlowHandle` which is a "cross-process" handle on a Flow object, and is used for termination.

- an_Flow an_flowCreate(an_Credentials *c, an_ThreadPoolSpec *tp, an_MemSpec *vm, an_NotifyFunc termHandler)
  See the Janos method `Flow.newFlow(Credentials cred, ThreadPoolSpec tPoolSpec, MemSpec memSpec, FlowStartSpec flowStartSpec, Runnable termHandler)`. Note that most of the parameters line up with the NodeOS C API. One difference in this case is the `FlowStartSpec`. This spec provides the set of Java class files that the new flow can see and gives the initial class and the set of arguments to pass to its `main` function. This deviation from the spec stems from a change to the ThreadPool that we are making (more below).
- void an_flowDestroy(an_Flow fid)
  See the Janos `Flow.destroy()` and `FlowHandle.destroy()` methods.
- an_Flow an_flowId(void)
  See the Janos `Flow.currentFlow()` method.

1.1.2 Thread Pool

Thread Pools are basically unchanged in the Janos implementation. We do expose a `Thread` object, but it contains only the features defined by the NodeOS Thread Pool interface. The Janos Java NodeOS bindings do not expose the ability to disable/enable interrupts or to change a thread's priority. Additionally, thread-interrupt handlers are not available at the Java level. Currently, the `interrupt()` method only provides the function provided by native Java; it will only wake sleeping threads or post an interrupt flag on them.

While the NodeOS interfaces are the same, one change is made to the semantics. The NodeOS C API maintains a strictly one-to-one binding between `ThreadPool` and `Flow`. We relax that restriction in Janos. Thus, in Janos, a flow's start function is part of the `Flow` object, and not part of the `ThreadPool` constructor (as in the NodeOS C API). This lets us create multiple `ThreadPools` within a single Flow, for example, to have different thread pools to service the control and data packets in a single logical flow.

- an_thread_t an_thread_self(void)
  See Janos `Thread.currentThread()`.

- an_error_t an_thread_start(an_thread_t *out_thread, an_Flow fid, void (*f)(void *arg), void *arg)
  See the Janos `ThreadPool.threadStart(Runnable r)` method.

- void an_thread_exit(void)
  Oops. We seem to have not implemented this one yet. Returning from the runnable that started the thread is equivalent, though.

- void an_thread_yield(void) See the Janos `Thread.yield()` method.

- void an_thread_sleep(const an_timespec_t *delay)
  See Janos `Thread.sleep(long millis, int nanos)`

- void an_thread_setdata(void *data)
  See Janos `Thread.setObject(Object o)`

- void * an_thread_getdata(void)
  See Janos `Thread.getObject()`

- void an_thread_interrupt(an_thread_t thread)
  See Janos `Thread.interrupt(Thread target)`. Beware, as none of the NodeOS-specified interrupt delivery or handling is yet really implemented in this Java binding. However, standard Java-specified "interrupt" is still available. I.e., this method will interrupt `wait()` calls. NodeOS methods protect themselves from interrupts and will delay the interrupt until after they have completed. Some long running methods will "correctly" handle an interrupt and return early.

- void an_thread_block_interrupts (void)
  Unavailable. See above.

- void an_thread_unblock_interrupts (void)
  Unavailable. See above.

- void an_thread_setprio(an_thread_t t, int priority)
  Not available at the Java level. The Java interface to thread scheduling parameters is not complete yet.

- int an_thread_getprio(an_thread_t t)
  Not available at the Java level.

In terms of synchronization, the Janos Java NodeOS interface provides the standard Java synchronization mechanisms in place of the NodeOS specified functions. Additionally, there is a `Semaphore` construct available. Mutexes may be added later, but we haven't yet had a need for them.

1.1.3 Channels

The Janos Java NodeOS interface wraps the low-level NodeOS Channel abstractions into three distinct classes: `OutChannel`, `InChannel` and `CutThroughChannel`. Instead of sending or receiving `Packets`, at the Java level `BufferHandle` objects are used.

Just as in the NodeOS spec, Java Channel objects are fairly static. The constructor determines the demultiplex key, address, protocol processing, and thread pool that the channel should use.

1.1.4 Memory Pool

All memory management in Janos is implemented within the JVM (err, it will be implemented in there). There are currently no memory management interfaces at the Java level except for specifying a limit when a new Flow is created (see `MemSpec`).

1.1.5 FileSystem

Janos does not define any wrappers for the POSIX-like file system interfaces specified in the NodeOS document. On the other hand, the existing JDK file interfaces are quite close to POSIX already. Janos does define some utility classes for logging, but they are "above" the NodeOS specifications.

1.1.6 Events

There is some support in the implementation of `ThreadPools` and their `WorkItems` to support events, but no public interface is yet implemented (ANTSR didn't need it...).

1.1.7 Heap

Janos uses the Java-standard `new` to allocate objects. Since Java is garbage collected, the reclamation of objects is hidden from Java.

1.1.8 Packets

The NodeOS defines a `Packet` object for sending and receiving chunks of data from the network. `Packets` are designed to simplify adding and re-writing headers. The Janos NodeOS interface provides a higher-level abstraction, the `BufferHandle` which, in addition to the services provided by a `Packet`, provide a level of indirection that makes Buffers suitable for sharing between flows. See the Javadoc description of `BufferHandle` for more detail.

1.1.9 Inter-flow Sharing

The Janos NodeOS interface is designed for a JVM (the JanosVM, actually) that will provide strong separation of flow heaps. As such, sharing between flows is strictly limited to raw buffers (shared via `BufferHandles`).

Additionally, a rudimentary inter-flow communication mechanism, the `CommSpace`, has been implemented. These interfaces and their semantics are not derived from any part of the NodeOS specification. They are unique to the Janos approach to multiple flows in a single JVM. Note also that at the Java level we do not plan to expose any of the inter-flow or inter-EE facilities specified in the NodeOS spec.

2 ANTSR

Within the ANTSR tree are three subtrees. Look at the documentation within each file to get a handle on its function. There are also a couple of `README` files scattered around that should help.

- `core/` — the core of the ANTS runtime. This is where most of the code to upgrade ANTS to the NodeOS abstractions is located.
- `apps/` — some of the ANTS demo applications ported to ANTSR. Ping, multicast, and mobile are ports of original ANTS demos. Pong, Stats and Statsfusion are all new demonstration applications.
- `route/` — the ANTS routing table and dynamic update algorithm. This is a reworking of Dave Wetherall's ANTS dynamic route application to ANTSR.

- `util/` — the standard ANTS utility directory.

For those who have an understanding of earlier versions of ANTS, the following high-level change list might be useful.

2.1 Changes to ANTS to make our ANTSR

2.1.1 Rationale and motivation

The NodeOS interfaces defines InChannels, OutChannels, and CutThroughChannels, Threads, packet filters, Addresses, and Flows (Processes). We take advantage of these primitives wherever possible in ANTSR. In many cases the new abstractions are much cleaner than the implicit abstractions defined in ANTS.

Many internal parts of ANTS had to change to accommodate precise resource control. For example, we start up a new Flow (a new "process") as early as possible in order to charge the resources used in downloading code and creating Flow-specific state to the appropriate Flow. This required fundamental changes to the way protocols are stored and forwarded.

We strove *not* to change the programming interface exported to ANTS protocols and applications. However, minor changes were introduced into the startup and definition of an application.

We updated class and object names. While making wholesale changes to the infrastructure we took the opportunity to rename classes that had evolved beyond their original design. For example, `Method` in the old ANTS actually stood for a Class. It is now named `CapsuleClass`.

We installed each protocol in its own namespace, using a ClassLoader. Each protocol instance now gets its own set of threads for handling dispatch, decoding and delivery.

2.1.2 Specific Changes

- Changed the package name; for example, `ants.Node` is now `utah.janos.antsr.core.Node`.
- Removed the `Entity`, `Manager`, and `ManagedObject` classes. These were for tracking logging and debugging state. This information has been moved out into the NodeOS logging interface and the NodeOS debugging interface. (See `java/io/Log*`, `java/core/Trace.Java`, and `java/core/Debug.Java`). The `Entity` class still exists; it simply does standardized argument parsing and usage printing.
- Dropped extensions for now. Not sure what they really are and have never seen one.
- The `NodeCache` only provides intra-flow state. There is no mechanism for sharing information between flows.
- Use specific exceptions where we can (replacing the generic Exceptions thrown all over).
- Made the ANTS Node.Java interfaces use the NodeOS Node.Java interfaces.
- Added the MIDID request/response as a stage in the dynamic code loading protocol. This provides information about the user and resource limits of the new flow that is being created.
- Made the distinction between an ANTS address (a 32-bit IP-like integer) and the underlying physical address (an IP address and port number) explicit and documented.
- Renamed `ANEPChannel` to `ANEPFlow`. Renamed `Method` to `CapsuleClass`. Renamed `InChannel` to `CapsuleChannel` (sort of). The "resource" limit on a Capsule is now merely a time-to-live hop-count.
- Moved the routing table and protocol out of the core ANTS package. The raw table lives in the NodeOS package, and the dynamic routing protocol is now just an ANTSR application. It is somewhat privileged (it gets to write to the route table).

- Take heavy advantage of the packet filtering functions in the NodeOS interface to directly dispatch packets to the appropriate `Flow`.
- Added a primitive "remote console" for managing a node via telnet. By default the console starts on port 2048. The `-consoleport n` parameter is understood by the Node configuration command. The remote console support a `info` command which reports information about various parts of the node, a `shutdown` command which shuts the node down, a `quit` command which shuts down the remote console connection and the `app` command which works just like the command in config files.
- Removed the dependency on AWT implicit in all of the basic ANTS classes (`Entity`, `Manager`, etc.). Applications will have to be updated to use the new `util.Frame` class.
- All of the classes are run through a preprocessor now, so it's simple to change the package name, make debugging-only changes, etc.

Credits

The development effort was led by Patrick Tullmann. The Java NodeOS layer was developed by Patrick and Chris Hawblitzel, while the latter was on a summer internship at Utah. ANTSR was developed by the same crew, building on earlier ANTS work by Godmar Back. Tim Stack and James Simister ported most of the ANTS applications to ANTSR and added many missing features to the ANTSR core. The original ANTS was developed by David Wetherall. Patrick wrote most of this document.