CS262 Fall 1998 Chang Lin / Jeff Tsay
December 15, 1998

x-kernel/J

1 Introduction

The x-kernel is a set of libraries and common interfaces that implement a framework for building protocol stacks. x-kernel protocols have has a number of advantages over the BSD socket interface, such as non-blocking I/O and extensibility. The x-kernel is fully described in [1]. The x-kernel was designed in an object-oriented fashion, but unfortunately the original implementation was written in C. In this paper, we will describe an implementation of the x-kernel written in Java, called x-kernel/J.

2 Motivation for a Java Implementation

Before we describe the implementation itself, we should examine how a Java implementation is an improvement over the C version, and how the x-kernel can be used to solve Java's networking problems.

Improvements Java brings to the x-kernel

1. Java is a real object-oriented language. Inheritance is part of the language, while in C, inheritance is somewhat achieved with careful layout of **struct**s. Because of the implicit **this** pointer, fewer arguments are required for each function (or method) call. In addition, member variables are implicitly de-referenced inside methods. This makes writing new protocols simpler.
2. Java has access protection on its members and methods, which prevents protocol writers from inadvertently modifying private members or calling private methods.
3. Java is platform-independent. Protocols written using the Java x-kernel will work on any platform that supports the Java Virtual Machine. Java as it is provides threads, synchronization, and timers, all of which have historically been non-portable from OS to OS. The original x-kernel contains machine dependent code, for all its supported platforms to implement these primitives.
4. Because linking in Java is done at runtime, protocol code can be downloaded and installed without rebooting or even recompilation. This allows Java x-kernel applications to be low-footprint "thin clients", which contain only bootstrap x-kernel code and download necessary protocols on the fly.
5. Java 2 provides data structures such stacks, hash tables, and vectors, that the C version implements hastily and includes in the core code of the x-kernel.
6. Java has its own garbage collector, eliminating the need for the one in the C-version.

Improvements the x-kernel brings to Java networking

1. Unlike the monolithic socket interface, the x-kernel is extensible. Protocols can built on top of any other protocol. In Java networking, only the UDP or TCP layer is visible to applications. Protocols may be built on top of UDP or TCP sockets but

there is not a standardized way of doing so. In addition, no protocols may built on top of the IP or Ethernet layer.

2. The socket interface blocks or spin-waits when receiving or sending data. This ties up many threads. In the x-kernel, threads almost never block, and they execute to completion quickly. The x-kernel paradigm is one-thread-per-message.
3. CPU time can be wasted if the `read()` or `write()` calls do not actually read or write any data, which is legal because `read()` and `write()` return a number of bytes actually read or written.

3 Changes from the Original x-kernel to x-kernel/J

Almost all of the functionality of the original x-kernel has been implemented in Java, including protocols, sessions, events, maps, messages, semaphores, tracing, assertions, and more. The unimplemented parts of the original x-kernel have to do with automatic configuration of the protocol and session graph, which must currently be done manually in Java. This is not a serious drawback because we expect the protocol and session graph to vary quite a bit, without restarting the x-kernel.

In addition, there are some utility routines made obsolete by Java or the Java 2 API :

Original routine	Java Replacement
`xMalloc()`	**`new`** operator
`xFree()`	Not needed
`xGetTime()`	`System.currentTimeMillis()`
`xAddTime(), xSubTime()`	Arithmetic on `longs`
`ipHostStr()`	`getHostAddress()`
`str2ipHost()`	`InetAddress.getByName()`
`xk_gethostbyname()`	`InetAddress.getByName()`
`Delay()`	`Thread.sleep()`

In the original x-kernel, all function calls operating on objects had a **this** pointer as their first argument. Since Java is an object-oriented language, all old-style calls like `func(p, ..)` have been replaced by methods like `p.func(...)`.

We will assume protocol programmers have access to the original x-kernel documentation of the libraries [2], and we only document the differences here.

3.1 The Superclass of all x-kernel Objects, `XKObject`

All objects in the Java x-kernel, with the exception of RecycleableThread, derive from `XKObject`, which provides the following methods, providing some of the tracing and panic functionality of the original x-kernel.

`public static final void` `assert(`**`boolean`** `condition, String errMsg)`
If the condition is false and `enableAssert` is true, throw an Error.

public static final void trace(String msg)
Print the message if enableTrace is true.

public static final void error(String msg)
Throw an Error.

public static final boolean acquireMasterLock()
Acquire the master lock. This is discussed in 3.2.1.

public static final void releaseMasterLock()
Release the master lock. This is discussed in 3.2.1.

public static final void shutdownAll()
Prepare for program exit by shutting down the event scheduler, and cleaning everything up. This should only be executed once by the main thread of a Java program.

Note that the assert() and trace() methods are not macros as in C, so the method arguments will always be evaluated. The static final boolean variables enableAssert and enableTrace, if set to false (requiring recompilation) may result in no code generation as in C, if the Java compiler is sufficiently sophisticated and the arguments do not have side effects.

XKObject also provides the following return values: XK_SUCCESS, XK_FAILURE, ERR_PROTL, ERR_SESSN, and ERR_ENABLE; and protocol control operation codes GETMYHOST, GETMYHOSTCOUNT, GETPEERHOST, GETPEERHOSTCOUNT, GETBCASTHOST, GETMAXPACKET, GETOPTPACKET, GETMYPROTO, GETPEERPROTO, RESOLVE, RRESOLVE, FREERESOURCES, and SETNONBLOCKINGIO which are documented in the original x-kernel documentation.

3.2 Encapsulating Pointers in Java : Pointer and its Subclass BytePointer

Java does not provide pointers to arbitrary locations within an array nor pointer arithmetic operations, so the customary way of accessing pointers as in C is to keep track of the base of an array as well as an integer offset into the array. This becomes quite cumbersome, especially when making method calls or returning the value of a "pointer" in a method, so the class Pointer and its subclasses BytePointer, IntPointer, etc. were written for the purpose of capturing the functionality of C pointers. These classes hide the fact that a pointer to an array and an offset into the array are actually used.

The customary operations that can be performed on or with pointers can be performed with Pointer and its subclasses :

Operation	C syntax	Pointer syntax
De-reference	*(p + i)	p.refAt(i)
Write to location	*(p + i) = z	p.setAt(i, z)
New incremented pointer	p + i	p.add(i)

Increment existing pointer	`p += i`	`p.addR(i)`
Copy to location at pointer	`memcpy(p1, p2, ...)`	`p1.copyInto(p2, ...)`
Null test	`if (p == NULL)`	`if (p.isNull())`

In addition, `Pointers` contain methods for comparing regions of memory.

In x-kernel/J, only `BytePointers` are currently used.

`Pointer` and its subclasses are provided in a separate package, `Pointer`.

3.3 Core Functionality - Protocols and Sessions

In the C implementation, callback function pointers in the protocol (`Protl`) and Session (`Sessn`) structures point to user code for the implementation of a protocol. In Java, a much cleaner way of specifying user protocol and sessions is used – users extend the `Protl` and `Sessn` objects and override the callback functions, which are :

For `Protl` :

```
protected int callDemuxCallback(Sessn lls, Msg request, Msg
reply)
protected int closeDoneCallback(Sessn session)
protected int controlProtlCallback(int opcode, BytePointer
buffer, int length)
protected int demuxCallback(Sessn lls, Msg message)
protected Sessn openCallback(Protl hlpType, Part[] participants)
protected int openDisableAllCallback(Protl hlp)
protected int openDisableCallback(Protl hlpType, Part[]
participants)
protected int openDoneCallback(Protl llp, Sessn session, Protl
hlpType)
protected int openEnableCallback(Protl hlpType, Part[]
participants)
```

For `Sessn` :

```
protected int callCallback(Msg request, Msg reply)
protected int callPopCallback(Sessn lls, Msg request, BytePointer
hdr, Msg reply)
protected int controlSessnCallback(int opcode, BytePointer
buffer, int length)
protected int duplicateCallback()
protected Part[] getParticipantsCallback()
protected int popCallback(Sessn lls, Msg message, BytePointer
hdr)
protected int pushCallback(Msg message)
```

These callbacks correspond one-to-one with the "protocol specific" operations referred to by the x-kernel user's manual, with the eliminated first argument, `self`, replaced with the implicit **`this`** pointer.

The protocol independent (or "generic") calls `xOpen()`, `xOpenEnable()`, etc. are public, final methods which do error checking and reference counting (which is still used in some parts of the implementation).

`xCreateProtl()` and `xCreateSessn()` have been replaced by constructors. The user must provide constructors for the objects derived from `Protl` and `Sessn`, call the superclass's constructor appropriately, using a **`super`**`()` call, then do any protocol-specific initialization in the body of the constructor.

The helper functions that default implementations use (`defaultOpenEnable()`, `defaultOpenDisable()` etc.) are provided as methods in the `Protl` class for use by protocol implementers.

The `numDown` field for a `Protl` has been replaced by a public method, `numDown()`, due to Java implementation using Java's `Vector` class.

3.4.1 Single Thread in xkernel/J at a Time

The original x-kernel documentation states that only one thread should be executing x-kernel code at one time. This is possible because in the x-kernel, threads typically do not block; instead they execute to completion. Thus the convention is that all threads must acquire a master lock before calling x-kernel code, and release it after completion. The documentation for x-kernel 3.3 mentions that there is a research group working on finer grained locks but we haven't heard anything more than that. In x-kernel/J we have reduced the problem a bit by making some libraries thread safe (e.g. Messages), but we still need the same locking mechanism for code that executes any `Protl` or `Sessn` code. Applications and device drivers must use the `acquireMasterLock()` and `releaseMasterLock()`, defined in `XKObject`, before making any calls to `Protl` and `Sessn` objects.

In practice, protocol implementers do not have to worry about this problem because by the time callbacks are called, the master lock has already been acquired. In addition, events automatically acquire the master lock.

3.4.2 Configuration of a Protocol Graph

The protocol graph is currently created by application code, instead of using a protocol graph file as in the original implementation. The application which creates the `Protl` objects links them together by calling `linkProtl`, defined in `XKObject`.

3.5 Message Library

3.5.1 External View of Messages

From the user's point of view, the Message library, implemented by the `Msg` object, is almost exactly the same as in the original x-kernel. The exceptions are that the `msgConstruct…()` functions are replaced by constructors, and that `BytePointer` objects are returned instead of real C pointers. In addition, since Java is garbage collected and even the `finalize()` method is not guaranteed to be called, it is no longer possible to assign freeing functions to each message.

3.5.2 Message Implementation Changes

The code for messages was for the most part converted line-by-line from the original C code. Messages actually are composed of many different types of data structures linked together, so as to avoid copying. The major substantive difference is that language level inheritance was used instead of careful structure layout in C. Thus all pointer conversions become legal.

Due to the interface change that no free functions are associated with messages, the Java implementation of Messages requires no "buffer nodes", which had an explicit destructor function associated with them.

3.6 Participant Library

The participant library creates stacks of `Address` objects, which can (and should) be subclassed by protocol implementations to contain useful address representations. The participant stacks can be accessed with:

```
public void push(Address addr);
public Address pop();
public Address peek();
```

An array of participant stacks, which represents a sender/receiver pair, can also be created, with the `initPartList()` method.

3.7 Event Library

3.7.1 Construction and Scheduling

Events, encapsulated by the class `XKEvent`, can be constructed in two ways :

`public XKEvent()`, or
`public XKEvent(EventFunctor ef, Object arg0, long usec)`

The first version creates an empty event, which can be scheduled later with the `schedule()` method (which has the same parameters as the second constructor). The constructor schedules the event immediately.

3.7.2 How to Execute a User-Defined Routine

The original x-kernel allows a user-supplied function, passed as an argument, to be executed after the specified delay. In Java, however, there no function pointers, so instead an object that implements the `EventFunctor` interface is passed to `XKEvent()` or `schedule()`. Objects implementing the `EventFunctor` interface implement only the single method, similar to the prototype used in the original x-kernel :

```
void f(XKEvent event, Object obj),
```

`which is executed after the specified delay.`

3.7.3 Timing Granularity

Unfortunately in Java, the system timer (the value of which is returned by `System.currentTimeMillis()`) only has a granularity of 1 ms, although microseconds are specified in scheduling events. Therefore, events will be executed with an average error time of 500 microseconds.

3.7.4 Event Library Implementation Description

Events are implemented by a single thread that monitors a queue of pending events. Each event is marked with a system time at which the user-supplied routine should be executed. The monitor thread peeks at the front of the queue (sorted by scheduled system time). If the event at the front of the queue is due, it grabs a thread from the thread pool (described later) to execute the event, and removes it from the queue. If the event at the front of the queue is not due yet, the thread waits until the event is due, or it is signaled that an event has been added to the queue. Thus events can be scheduled out of order.

The event scheduler maintains separate queues for pending, cancelled, running, and finished events. Most event operations are simply moving events between these queues. The queues are implemented by Java 2's `TreeSet` class which does the sorting of the events based on the `compare()` method. The `compare()` method, implementing the interface `Comparable`, simply uses the scheduled time to order 2 events.

When an event is run, the master lock must first be acquired, and when the event is finished, the lock must be released. This is done invisibly to the protocol programmer, so he need not worry about the lock when implementing `f()`.

Threads that run the event are taken from a thread pool. The thread pool contains threads (of class `RecycleableThread`) which wait until signaled by the pool manager, execute the run() method, and then notify the pool manager that it is returning to the pool. Then they repeat the same process until the system is shut down. The thread pool manager manages two linked lists of `RecycleableThreads`: those that are running and those that are waiting for another job. Dispatching a thread consists of grabbing a thread from the waiting list, putting it on the running list, and notifying the thread. When a thread returns to the pool, the reverse process is initiated.

The thread pool must first "warm up" by creating `RecycleableThreads` until the pool is full, and then it can start to reuse these threads. If there are no threads ready to service a thread request, the pool manager just creates a normal thread, which does not return to the pool.

The purpose of the thread pool was to remove the overhead of creating and destroying `Thread` objects. Since threads in x-kernel never block and execute for short periods, a lot of threads would need to have been created and destroyed if not for the thread pool.

3.8 Map Library

3.8.1 Two Types of Maps : `XObjMap` and `XMap`

While the original version of x-kernel had a single map library, the xkernel/J requires two types of maps : one type that maps `Objects` to `Objects` and one type that maps contents pointed to by `BytePointers` to `Objects`. The reason two types are necessary is that in Java, pointers cannot be converted from one type to another, and so two `Objects` cannot be compared byte-by-byte as in C. Instead, the `equals()` methods of `Objects` are used to test for equality, in an `XObjMap`, while the contents pointed to be `BytePointers` are compared byte-by-byte, in an `XMap`.

If one wishes to map `Objects` to `Objects` with the `XObjMap` class, the key `Object` should have its `equals()` method overloaded, to do member-wise comparison (unlike the default `equals()` method which only compares the pointers to the `Objects`).

The constructor for `XMap`, which maps byte strings to Objects, requires a key size, in bytes, as an argument. If a byte string with a length longer than the specified key size is used to lookup in the map, it is converted internally to a string with the same length without any actual copying.

3.8.2 Resolution Syntax Change

The resolution (or lookup) methods of both types of maps have a syntax slightly different from the one in the original x-kernel, which had the following prototype :

```
XkReturn mapResolve(Map map, void *key, void **id);
```

The original function would store a pointer to the map's value at `*id`, and return an error code. In Java, without the `&` address of operator, this is cumbersome, and so the function has been replaced with the following methods :

`public Object resolve(Object key)` for `XObjMap` and,
`public Object resolve(BytePointer key)` for `XMap`

These methods return the map value on success, and **`null`** on failure.

3.8.3 How to Use `forEach()`

The original x-kernel allows a user-supplied function, passed as an argument, to operate on all entries of a map. In Java, however, there no function pointers, so instead an object that implements the `MapFunc` interface is passed to `forEach()`. Objects implementing the `MapFunc` interface implement only the single method :

```
public int map(Object key, Object id, Object arg);
```

which is basically the same as the prototype for the original x-kernel's user-defined functions. The return values for user-defined methods (`MFE_FOUND`, `MFE_CONTINUE`, `MFE_REMOVE`) are contained in the `MapFunc` interface.

3.8.4 Map Implementation Description

Both types of maps use Java hashtables to map keys to values, thus the dependence on the `equals()` method of objects. XMap relies heavily on the methods provided by BytePointer to manage byte strings and compare them.

3.9 Thread library

The original x-kernel provided a "thread library" that is actually just a semaphore implementation and a thread delay function. These should not be used frequently, as normal x-kernel threads do not block. The exception the documentation gives is waiting for a reply from an RPC protocol. In xkernel/J, semaphores have been implemented in class `Semaphore`, and thread delay can gotten from `Thread.sleep()`.

3.10 Utility library

3.10.1 Byte Ordering / Data Insertion

Since Java does not have network byte ordering functions, x-kernel/J contains various utility functions to insert and retrieve ints, shorts, and byte strings to and from a given array, presumably a header. These resize in the base XKObject class.

```
public static final void putShort(byte[] packet, short s,
      int offset);
public static final void putInt(byte[] packet, int i,
      int offset);
public static final void putBytes(byte packet[], byte data[],
      int offset);
public static final byte[] getBytes(byte packet[], int offset,
      int len);
public static final short getShort(byte[] packet, int offset);
```

```
public static final int getInt(byte[] packet, int offset);
```

3.10.2 Checksum

```
public static short checksum(byte header[]);
```

The checksum function implemented in x-kernel/J is the same as the IP checksum. It is the negated 1's complement sum of the input data read as network byte ordered (big-endian) words. It resides in the class `OCSum`.

4. Protocol Implementation with x-kernel/J

We implemented functional versions of the UDP, IP and Ethernet protocols. We will not describe the complete details of their operation here, but only point out facets of their operation which are new to our design.

1. Since the Java API contains no methods for accessing Ethernet hardware, we wrote a set of Java native methods in C++ to perform this task. These methods use Linux packet sockets to send and receive data from the Ethernet hardware.

2. Every message that is received from the network causes a new thread to be spawned to handle the message's path up the protocol graph. To reduce the overhead caused by Java's medium-weight thread creation, we recycle threads using a thread pool, and reuse them by switching their run code.

3. The application program must contain a subclass of `Protl`; the top-level protocol will expect to be linked to a `Protl` above it, and will pass messages to the application by calling the `Demux()` method of this `Protl`. The `Demux()` method should only place the incoming messages in a queue and return. This insures that thread lifetimes for incoming messages will be short; they will be created by the network driver, and be destroyed immediately after they reach the application layer. The blocking or non-blocking receives that the application might wish to implement should be executed in a separate thread which removes messages from the application's `Demux()` queue.

Although we felt that Java was a better fit for the x-kernel design than C, it is not without its problems. Its current incarnations were not designed for systems programming of this nature. All types are signed, which necessitates the use of unnecessary shift operations to emulate unsigned byte types for headers and to reorder words into network byte order. Also, in C-derived languages, headers can be created easily by filling in fields in structs and "flattened" into byte strings by simply casting the struct pointer to a character pointer. Java has no such "flattening" operation, so we created headers as flat byte strings to begin with and then wrote various methods to painstakingly insert data in the byte strings in the correct network byte order. Also, network code is performance-critical, which is slightly at odds with Java's machine-independent byte code paradigm.

When sending and receiving 256-byte packets, our current UDP/IP/Ethernet "mini-stack" adds an average of 2ms over normal Java socket call time. For receives, this overhead is due to copying incoming packets into messages and garbage collecting them when the acceptor thread expires. For sends, the overhead is mostly due to inefficient byte manipulations when creating headers, and partly due to DNS and ARP queries.

References

[1] Hutchinson and Peterson. “The x-Kernel: An Architecture for Implementing Network Protocols”.

[2] Network Systems Research Group. “x-kernel Programmer’s Manual”. June 1997. (Available from http://www.cs.arizona.edu/xkernel/).

[3] Network Systems Research Group. x-kernel source code, v.3.3. October, 1998. (Available as ftp://ftp.cs.arizona.edu/xkernel/xkernel.tar.Z).

[4] Postel, J. (editor) “Internet Protocol – DARPA Internet Program Protocol Specification”, RFC 791, Information Sciences Institute, September 1981.

[5] Postel, J. “User Datagram Protocol”, RFC 768, Information Sciences Institute, August 1980.