Originally implemented to support wireless sensor networks, the lightweight mobile code daemons described in this article now work on wired networks as well. The authors have implemented them to run on multiple operating systems and architectures. The daemons use a peer-to-peer indexing scheme to find mobile code packages as needed. Calls to the daemons are automatically dereferenced to find the proper package for a node’s OS and hardware.

Advances in network and embedded processors make possible entirely new classes of distributed applications. These applications must not be vulnerable to failures of single nodes or links, and the networks must be able to tolerate intrusion and the subversion of a small number of nodes. As part of a research agenda to create a secure adaptive infrastructure, the lead author and his students have implemented systems combining mobile code and peer-to-peer networking. Thus far, we have implemented applications for sensor networks, adaptation to denial-of-service (DoS) attacks, dynamic battle management, and building-control systems (BACNet).

The mobile code daemon we present is based on a core network protocol called the Remote Execution and Action Protocol (REAP), which is responsible for passing messages between nodes within our network. On top of this packet protocol, we have developed a framework that allows objects to serialize themselves and travel across the network. At a higher abstraction layer, we’ve written messages to handle remote process creation and monitoring, simple file-system operations, and resource-index operations.

We’ve used the REAP daemon as middleware for creating robust distributed infrastructures for applications interacting with chaotic environments. The first application is a battery-powered wireless sensor network in which nodes are fielded with a minimal software suite that is extended as needed by using mobile code. Classification methods assign targets to classes (trucks, motorcycles, and SUVs, for example) as they are detected; each classification method performs differently on various combinations of targets. In our implementation, nodes automatically determine
which classification program is best suited to the current mix of targets and download the ones they need. Given that nodes in a given region will have the same mix of targets, programs usually travel a small number of hops during a download. The daemons allow nodes to easily reconfigure their local software configurations dynamically.

The second application is a dynamic battle-management scenario. A distributed process, such as target allocation, consists of multiple tasks that might be performed at different locations. The enemy will attempt to disrupt the target-allocation process by using DoS attacks. The system uses the mobile code daemon for introspection (locating processes that become isolated from the network) and reallocates tasks to new nodes as required to replace processes that have been isolated from the global system by a DoS attack. We are currently deriving the game-theoretic strategies needed to ensure that task allocations minimize the likelihood of future disruptions. Once again, our mobile code daemons provide the infrastructure needed for adapting to a chaotic environment. In this case, they let the network reconfigure its global topology dynamically.

This article describes the daemon’s internal structure and its interactions with the global system. This small layer of middleware enables the system to adapt to attacks and changes in its environment. We then discuss the daemon’s global structure, including detailed descriptions of individual components.

Mobile Code Paradigms

Our mobile code daemon framework is built on the taxonomy of existing mobile code paradigms shown in Figure 1. Each paradigm places constraints on its systems’ behavior. In the taxonomy, a transmission is a set of messages between threads on hosts. A system’s behavior is defined as the itineraries its transmissions follow. An itinerary is the sequence of nodes visited by a mobile code package. Figure 2 shows the definition of a message: each has an instruction that defines some action to be taken and a payload that specifies the action’s (possibly empty) target. In this model, resources, threads, and programs can be fixed or mobile.

This taxonomy shows that the established mobile code paradigms of client–server, code-on-demand, remote evaluation, and mobile agents can all be expressed using a single abstraction. A mobile code system is defined by its distributed behavior – the set of message transmissions that follow an itinerary between cooperating nodes. Messages can consist of programs, data, execution commands, resource-allocation requests, and so on.

Existing paradigms and mobile code implementations are all limited instances of this taxonomy. Code-on-demand, for example, is limited to code requests moving from the initiating host to the target, followed by code-migrate messages in return. As another example, mobile agents are series of code (and state) migration requests in which the agent determines the itinerary. We’ve used this taxonomy as the basis of an API for developing a flexible mobile code execution environment. To cooperate, each node must offer a local execution environment that remote nodes can access.

The daemon presented here provides a com-
mon execution environment that’s somewhat analogous to a Java Virtual Machine. Unlike the JVM, however, REAP can mimic any of the established mobile code approaches. It can also create applications that don’t fit the common paradigms.

Architecture

The REAP daemon is written in C++. The first version ran on the Windows NT and CE operating systems; we have since ported it to Linux as well. As Figure 3 shows, the daemon is structured into several core modules: foundation classes, networking core, random graph module, messaging core, packet router, index server, transaction manager, resource manager, and process manager.

Foundation Classes

Before discussing the REAP daemon in detail, we should describe the underlying framework on which it is built. The framework abstracts many of the systems-programming complexities out of the core and into a set of libraries. Thus, we’ve written our own object-oriented threading and locking classes, whose current implementation calls into the underlying operating system’s threads library. We also rely heavily on a set of templated, multithreaded, linked-list, hash, and heap objects throughout the code. In addition, there are classes to handle singleton objects (objects with a single instance), the union-find problem (locating sets with common components and combining the sets), and object serialization (packaging objects for transmission over the network). Lastly, a polymorphic socket library also allows different networking architectures to emulate unicast stream sockets, regardless of the underlying network protocol or topology.

The Networking Core

The daemon can communicate over several networking technologies. The major ones are TCP/IP, Diffusion Routing, and Unix domain sockets, but the socket framework’s design makes it easy to insert new protocols into the daemon. To achieve this, an abstract base class, Socket, includes all the familiar calls to handle network I/O. Furthermore, every node either has, a priori, a protocol-independent unique address or access to an algorithm for defining one. To open a new socket, the daemon looks up a node’s network-layer address in a local cache, and then opens the lower-level socket. When a cache miss occurs, a higher-level protocol uses the daemon to locate the network-layer address. The daemon allocates the appropriate socket on the basis of the destination’s network-layer address. Diffusion provided some interesting challenges because it is not a stream-oriented unicast protocol. Rather, it is essentially a multicast datagram protocol that provides a publish-and-subscribe interface. Thus, we had the choice of rewriting the REAP socket protocol as a datagram protocol or building a reliable stream protocol on top of the Diffusion framework, which we deemed the simpler option.

In essence, we wrote a simplified userspace TCP stack. It employs the standard three-way handshake protocols for opening and closing sockets, as well as a simple delayed-ACK algorithm. We implemented this system as an abstract child of the Socket base class. Our Diffusion driver implements our userspace TCP module, performing a role equivalent to the IP-layer processing code in most kernels: it receives datagrams from the Diffusion daemon through callback functions, parses the headers to make sure the datagram has reached the correct destination, and then either discards or passes the contents up to the TCP layer. We felt these steps were necessary because Diffusion is a multicast protocol, which means we can’t rule out the possibility that datagrams that weren’t destined for our socket object might reach it.

Random Graph Module

Early on in the project, we realized that persistent connections between the various nodes were essential. A single file transfer of a shared object could result in thousands of packets traversing the network, and session setup time was simply too
long over TCP and Diffusion. To counteract this problem, we implemented a system in which sockets are kept open whenever possible. Our first implementation opened directly to a destination and didn’t support multihop routing very well. In the current implementation, we use socket time-out counters to close underutilized sockets, but this method has inherent scalability problems.

To solve the scalability issues in the next version, we plan to use a multihop packet-routing network built on top of a random graph of sensor nodes. Each node in the system has four graph parameters:

- minimum degree,
- maximum degree,
- cliquishness, and
- clique radius.

The cliquishness parameter defines the probability that a new edge will be formed to a node within the clique radius. The minimum- and maximum-degree parameters control how many neighboring nodes can exist at any point in time. The clique parameters let us control the size and connectedness of cliques within the graph. (Clique — highly connected subgraphs — become more important when we investigate the index system.)

To add a new edge, the daemon generates a random number to decide whether or not to add a clique edge. It then chooses a random node from the node cache according to two filter criteria: the chosen node must have a minimum path length of two to this node, and its minimum path length must be less than or equal to the clique radius for a clique edge or greater than the clique radius for a non-clique edge.

Figure 4 shows example graph structures that would be embedded into the distributed system.

### Messaging Core

The messaging system implements REAP’s core. At the lowest level is a packet protocol, on top of which the daemon builds serialized objects. The Packet class is just a variable-sized opaque data carrier that can send itself between nodes; it also performs data and header checksumming. Figure 5 illustrates the REAP packet layout. The header defines enough information to route packets, specify the upper-level protocol, and handle multipacket transmissions for which the number of packets is known a priori. The options field consists of a 4-bit options vector and a 4-bit header-extension size parameter. The TTL (time to live) field is used in the new multihop protocol to eventually destroy any packet-routing loops that might form.

Higher-level messaging functionality is handled by a base message class and a set of classes that do object serialization. The serialization class in REAP provides a fast method of changing common data types into network byte-ordered, opaque data. This serialization system’s key advantage is that it handles only common data types, which means it has much lower overhead than technologies such as External Data Representation (XDR) or ASN.1.

The base messaging class provides a simple interface for controlling destination address,
source-transaction information, possible system-state dependencies for message delivery, and message sending. In addition, it defines the abstract serialization and reordering functions implemented by all message types.

The serialization class sits beneath the base message class and does the physical work of serializing data, packetizing the serialized buffer, and then injecting those packets into the router.

An object-serialization class receives these packets and inserts them into the proper offset in the receive buffer. A union-find structure keeps track of packet-sequence numbers; once it detects that all packets have been received, it delivers the message to a message queue in the destination task structure, which we discuss further in the “Transaction Management” section.

The run function is another interesting feature of the messaging system. It takes a task structure as an argument and is generally intended to perform some action on the message’s destination.

**The Packet Router**

Among its key responsibilities, the daemon packet router has two primary tasks: using its internal routing tables to move packets from source to destination and coordinating the dissemination of multihop routing data.

The packet router determines multihop paths via broadcast query messages, gradually increasing broadcast TTL until it finds a route or reaches a TTL upper limit, at which point the node is assumed down. This methodology helps reduce flooding, while increasing the likelihood of locating optimal paths. A simple optimization allows any node to answer a multihop query if it has an answer in its routing table. Although this system is essentially a heuristic, it tends to work well because neighboring nodes can easily bypass failed intermediate nodes when they find they can’t reach the next hop. Of course, this can lead to much longer paths through the graph, but support is integrated to warn of intermediate node failures as multihop cache expire times occasionally force refreshes. The multihop refreshes are performed in a unicast fashion; a broadcast refresh is used only if the daemon detects a significant hop-count increase.

The actual routing of packets involves the two destination fields in the packet header. First, the router checks whether the destination-node identifier is equivalent to the current node’s identifier, the local loopback address, or one of several addresses defined for special purposes (broadcasting to all members of a clique, for example). The router then checks whether the destination process identifier is equivalent to that of the current process. If not, the packet must be forwarded across a Unix domain socket. If both tests are passed, the packet must be delivered to its destination task. Because packets contain insufficient routing data for delivering them to a specific task, we must recreate the high-level message object in the router to determine the message’s final destination.

Every task in a REAP process registers itself with the router during initialization. Once a task is registered, it can receive messages bound for any active request. Several special tickets are defined for every task that handles task-status messages or task-wide requests. Other tickets are ephemeral and are allocated as needed.

**The Index System**

An important component of the REAP daemon is the index system, which implements a distributed database of resources available on the network. Each record in this database describes an object of one of the following types: index server, file, executable, library, pipe, memory map, host, or task. Every record in the database is associated with a canonical name and resource locator — both stored as human-readable strings as well as metadata to facilitate data and metadata replication.

Our goal is to establish a distributed cluster of index servers that transparently replicate each other’s index records, and a resource-control system that transparently replicates the actual data as well. At this point, however, the replication technology is only partially implemented.

The index system comprises four modules: client, server, database, and associated messaging protocol. The client is responsible for building a query message, sending that message, and awaiting a response or, with asynchronous calls, returning a response handle to the client. The index server is nothing more than a pool of threads that accept a certain type of message and then allow the messages to perform their actions. (In this sense, the REAP messaging system implements the mobile agent paradigm.) The server threads poll for incoming messages on the server task structure. When a thread receives a message, it runs the query embedded in the message against the local database and then sends the results back to the client in a query-result message.

The query system is based on a fairly extensible parse tree. Figure 6 illustrates our query language’s context-free grammar. It permits complex Boolean
filtering on nearly any variable defined in an index record. The index server is essentially a lightweight SQL server tailored to resource location.

The index infrastructure is mainly built on two message types: query and result. The query message consists of an operand tree, some query option flags, and possibly a list of index records. Once the query message reaches the server, a server thread receives it and calls the run function, which performs a query against the index database object and sends a result message to the source node. Once these actions are complete, the run function returns and the index server deallocates the query object.

The index system’s other major feature is that it lets the request select code based on a destination system’s architecture and operating system. Every index record contains an enumeration defining its membership in each hierarchy. When a node requests object code or binary data, we must ensure that it is compatible with the destination node.

When a query indicates architecture or operating-system concerns, the daemon makes C++ dynamic cast calls to ensure compatibility. Because we use C++ dynamic-casting technology, supported architectures and operating systems are determined at compile time. It wouldn’t be a technically difficult modification to use human-readable strings and runtime-defined polymorphic hierarchies, but we chose the compile-time approach because it is faster – especially with the relatively constant architectures and operating systems in our lab.

As an example of how this technology works, let’s consider a sensor node with raw time-series data that needs to be run through a fast Fourier transform (FFT). Suppose a distributed process scheduler determined that it would be optimal to move the raw data to a wireless laptop deployed in the field. When the laptop ran the FFT, it would query the index database for a given FFT algorithm and request architecture polymorphic checking. Let’s say this laptop had a processor with Intel’s SSE and MMX extensions, but not the SSE2 extensions. When the index server processed the query, it might find FFT algorithms that were compiled for 386, Pentium, SSE, Pentium 4, and Alpha EV5 processors. Filtering these queries, the server would determine that it could cast the laptop into 386, Pentium, and SSE, but not Pentium 4 or Alpha EV5. The laptop would then attempt to download the optimal mobile code package, dropping to slower implementations only when it couldn’t download the fastest.

Figure 6. Index query context-free grammar. Our query language permits complex Boolean filtering on nearly any variable defined in an index record.

Transaction Management
Every operation in REAP is addressed via a transaction address, consisting of a 4-tuple (node, process, task, ticket). These globally unique addresses enable flexible packet routing. One of our major goals was to facilitate network-wide interprocess communication through a simple high-level interface, without introducing high overhead.

Supporting the complex transaction-routing system requires a task-control structure. In our implementation, all threads and other major tasks have their own task structures, which are registered with local packet routers and define where message structures get delivered. The task structure’s primary jobs are to handle message I/O and allocate tickets. (A ticket is a token.) Because every active ticket has
an associated incoming message queue, our framework makes it possible to receive messages for specific tickets. As an added feature, REAP supports message-type filtering at the task level: the daemon deallocates any messages that fail to pass the filter rather than delivering them to the task.

The transaction-management system also handles task monitoring. We employ a publish–subscribe model that lets any task request status information from any other by subscribing to its status information service, which publishes status messages that the subscribed task will receive. Currently, all status information is sent as unicast datagrams.

The system’s main purpose is to notify the requester that its request has been received, and to notify it again when the request is completed. Another interesting application of this technology could include distributed process schedulers that monitor progress and system load on a cluster of nodes, and then schedule compute jobs to distribute the load to meet predefined criteria.

**Resource Management**

The resource-management framework is tightly coupled with the index system: when a client program wants to access a resource, it queries the index system. The daemon then passes the results to the resource-management object. Next, the resource manager attempts to open a resource from the result set. If possible, it will open one from each canonical name in the result set. The resource manager can thus overcome node failures by looking for other copies of a given resource. The current implementation attempts to open one instance of every canonical name in parallel, continuing this iterative process as timeouts occur. Eventually, the resource manager will open an instance of every canonical name or run out of instances of a resource in the index result set.

The resource-control system is built on top of a client–server framework, which we chose because the types of resources we want to support are generally not concurrent objects. Thus, the resource-management system consists of two REAP message types: resource operation and resource response. There are also two types of resource objects: client and server. For any given resource, there will be exactly one server object and one client object per task with an open handle to the resource. When a given client wants to perform an operation on the resource, it will send a resource-operation message to the server object’s transaction address. The server will then call the message’s run method and, through a set of polymorphic calls, perform I/O operations on the server object. The daemon will then send a response message to the originating node.

The client and server resource objects are based on an abstract interface that defines several common methods that can be used on Unix file descriptors. The major base operations are open, close, read, write, lock, unlock, read, write, and stat. In all cases, REAP provides blocking and nonblocking versions of these functions (the blocking functions are simply built on top of the nonblocking code).

As a simple performance improvement, we constructed client and server caching objects that perform both data and metadata caching. Given that our distributed resource interface is essentially identical to the virtual filesystem interface that Unix-like kernels give to applications, standard locking semantics can apply. Thus, our caching module simply looks at the file’s open-read and write modes’ file descriptors to determine an acceptable caching strategy. For the multiple-readers and single-writer cases, we allow client-side caching. For all others, we must disable client-side caching to avoid data inconsistencies. Thus, our caching semantics are identical to those used in the Sprite Network Filesystem.7 The REAP framework makes our implementation very simple because our mobile-agent-based messages can easily turn client caches on and off with minimal overhead.

To demonstrate this resource-control model’s power, we built client and server objects to support a distributed shared-memory architecture—again employing the abstract client–server caching model to increase performance.

**Process Management**

The last major component of the REAP framework is process creation and management, which consists almost entirely of message types. The primary type is the process-creation message, which contains both an index record pointing to the binary to execute and the argument and environment vectors to include. A second type is the process-creation response message, which simply contains the newly created process’s transaction address. Finally, the daemon can use task-monitoring messages to track the progress of a task using the publish–subscribe model described earlier.

**The Daemon in Practice**

To illustrate how our mobile code daemon middleware is used in practice, we provide two simple pseudocode examples that represent the two major...
families of mobile code paradigms: client–server and mobile agent.5

For remote evaluation – a member of the client–server mobile code family used by Corba factories and SOAP – we use the following pseudocode:

```pseudocode
//Locate appropriate host
Host = IndexQuery(Service_name);
//Construct appropriate data inputs if needed
Payload=ConstructPayloadMessage();
//Transfer program (named code) and data to remote host
// and wait for response
Response=RunCommandWait(Host,Code, Payload);
```

As a result, local thread (X) transmits three concatenated messages to remote thread (Y). A message containing the executable code is concatenated to a client–server-style transmission. After execution, Y sends possibly null execution results to X.

The mobile-agent paradigm uses two threads for each hop. We accomplish this with the following code:

```pseudocode
//Do local processing and collect data
Payload = LocalCommands();
//Locate appropriate host for next hop
Host = IndexQuery(Service_name);
//Move agent (named code) to next hop and execute.
RunCommand(Host,Code,Payload);
```

Thread X executes locally and composes a thread-migrate message containing agent code and state. This message is transmitted to thread Y on the remote host, where execution continues. A set of n hops requires n transmissions between up to n + 1 threads. The agent decides when and where to migrate.

Conclusions
The REAP mobile code daemon lets us experiment with many different mobile code paradigms over a fault-tolerant multiprocess framework. Because it provides a simple cross-platform, distributed interprocess communication framework, it is very useful for developing systems of collaborating distributed processes. This approach can mimic all the major mobile code paradigms,5 and its polymorphic code-selection system lets us use the optimal algorithm on a given system without significant user interaction. Finally, the distributed resource-management system allows us to reduce bandwidth requirements and permit concurrent use of resources without breaking normal concurrency rules. In future work, we plan to integrate reconfiguration strategies into the daemon.

Acknowledgments
The authors’ work is supported by the US Office of Naval Research (ONR) under award number N00014-01-1-0859. The opinions, findings, conclusions, and recommendations expressed in this article are those of the authors and do not necessarily reflect the ONR’s views.

References

Richard R. Brooks is an associate professor of electrical and computer engineering at Clemson University. His research interests include network security, sensor networks, and self-organizing systems. Brooks received a BA in mathematical sciences from the Johns Hopkins University and a PhD in computer science from Louisiana State University. He is a senior member of the IEEE. Contact him at rrb@acm.org.

Thomas E. Keiser is an undergraduate student in computer science and electrical engineering at The Pennsylvania State University. His research interests include network programming and computer security. Contact him at tkeiser@psu.edu.