Communication with Super Flexible Messaging

Alexey S. Fedosov and Gregory D. Benson
Department of Computer Science
University of San Francisco
San Francisco, CA, USA
{fedosov, benson}@cs.usfca.edu

Abstract *Super Flexible Messaging (SFM) provides a powerful and elegant message passing abstraction for transferring arbitrary data between remote processes. SFM achieves the simplicity of stream-oriented mechanisms and the ability to transfer structured data as found in high-level remote object invocation systems. Leveraging off of dynamic typing and named arguments in functions, SFM offers a simple syntax for sending structured data as attribute-value pairs between processes. A programmer can easily concentrate on algorithmic development because the specification of data to be transferred is done exclusively using a* `send()` *invocation. The* `recv()` *call can selectively choose incoming messages by specifying concrete attribute-value pairs or by constraining matches to a subset of attribute values. Our current implementation is in Python, but the SFM concept can be ported to other dynamically typed languages with named argument syntax. This paper presents the SFM mechanism, its implementation, some SFM examples, and a performance evaluation.*

Keywords: Python, message passing, communication

1 Introduction

Interfaces for data communication are crucial to the development and execution of distributed-memory programs. As such, a multitude of communication mechanisms is available to programmers. The ubiquitous sockets interface for TCP/IP [14] is provided by most operating systems and exposed as a higher-level module in most programming languages (e.g., the standard Python `socket` module [2] and the Java `net.Socket` class [7]). Higher-level middleware mechanisms such as remote procedure call and remote method invocation are built on top of a sockets interface. Such mechanisms include Sun RPC, Java RMI, CORBA remote objects, SOAP, and XML-RPC [4]. In high-performance computing, the Message Passing Interface (MPI) [10] provides a standard set of mechanisms for point-to-point and collective communication. Finally, several parallel and distributed programming languages incorporate dedicated syntax and semantics for achieving communication between remote processes [1].

All existing mechanisms are useful in appropriate contexts and they all have advantages and disadvantages. Our observation is that the low-level mechanisms provide simplicity in concept, but require detailed handling of data. Likewise, the higher-level mechanisms hide some of the complexities of the underlying low-level mechanism, but introduce new burdens on the programmer. For example, TCP sockets require a receiver to handle partial reads. Also, explicit serialization is required for sending structured data. In contrast, complete objects can be accessed using a remote object system such as Java RMI. Such systems require some form of interface definition, separate compilation tools, and a configured execution environment. Elegant mechanisms exist in dedicated distributed programming languages, but their usage is limited because their implementations are typically not widely supported and mainstream developers tend to use more general purpose programming languages.

We have developed a new communication mechanism called *Super Flexible Messaging* (SFM) that provides a simple way to send arbitrary, dynamically typed messages between remote processes. Like traditional low-level mechanisms, SFM allows for the explicit transfer of data between processes, and like higher-level mechanisms, SFM automatically serializes structured data. SFM handles all the low-level details and allows a programmer to send structured data without any additional interface specification. In SFM, the location in the source code where `send()` is invoked serves as the specification. The sender and receiver implicitly agree on the type of data transferred. SFM can be viewed as an extension of function invocation semantics in dynamically typed languages. In such languages, the caller and callee agree on the types of data passed in

and returned by convention. No static type checking is performed. Type errors are determined at run time.

The main operations in SFM are `send()` and `recv()`. The `send()` operation allows the programmer to specify an arbitrary list of attribute-value pairs. Each value can be an instance of a serializable type supported by the host language. The `recv()` operation allows the programmer to specify matching criteria also as attribute-value pairs. In addition, it is possible to match incoming messages by specifying a subset of attribute values. This form of message passing allows developers to focus on algorithmic rather than messaging details. It also encourages rapid development, because the amount and type of data to be transferred need only be changed at the `send()` and `recv()` call sites.

The current SFM prototype is implemented in the Python programming language [11]. We achieve a concise notation for SFM using Python's named argument syntax for function invocation. In this way, a complete SFM message is specified in a single `send()` call. SFM supports the transfer of all Python objects that can be serialized (pickled) including immutable types, lists, dictionaries, and user-defined objects. Received messages are returned as Python objects in which the object attributes are the names of the message attributes. Therefore, the syntax for accessing the received data is also very concise. The Python lambda type is used to constrain matching based on attribute values.

The development of SFM grew out of necessity in the development of a new Python research framework for distributed programming called River [12]. We wanted an efficient send/receive mechanism so we decided to build our underlying protocol on top of sockets. Prior experience in implementing MPI from scratch [5] led us to focus on developing a fixed message structure. However, we did not yet know what we wanted in the message structure. We concluded a lazy approach would be best to encourage rapid development, so we decided that a message could hold anything. To date, SFM has been used to develop the River prototype and several River extensions including an implementation of MPI and MapReduce [6]. The SFM concept has been extremely successful in facilitating rapid development of both distributed run-time systems and applications.

The rest of this paper is organized as follows. Section 2 describes the SFM mechanism and interfaces. Section 3 presents example SFM usage. Section 4 provides some details of our Python-based SFM implementation. Section 5 summarizes experimental results for communication with SFM. Section 6 talks about our experience using SFM in implementing the River framework. Section 7 makes some concluding remarks.

2 Mechanism

The SFM mechanism assumes a distributed execution environment in which processes can be named and discovered. SFM associates a receive queue with each named process. The SFM `send()` operation must explicitly specify a process name as the destination. Messages are deposited on the receive queue for a process. The `recv()` operation retrieves qualifying messages from the receive queue.

An SFM message is a set of attribute-value pairs. An attribute is always a string, whereas a value can be of any type, such as integer, string, list, object, etc. Two special attributes are reserved and have special meaning: `src` and `dest`. The `dest` attribute is required when sending and specifies the name of the destination process. The `src` attribute is added automatically by the SFM system and contains the name of the originating process. If the programmer leaves out `dest` or attempts to provide `src` in a `send()` operation, an exception is raised. Thus, sending a message can be as simple as passing all the desired attributes and their values as named parameters to the send function. The syntax for `send()` is:

```
send(dest=<process> [, <attribute>=<value> ] ...)
```

Note that it is possible to send an empty message for synchronization purposes. Typically, at least one attribute-value pair is provided in addition to the `dest=<process>` pair. Consider the following example:

```
send(dest='B', tag='input', data=[1,2,3,4,5])
```

Note that the attribute names are completely arbitrary. Also, data can come from the surrounding scope:

```
newdest = 'p1'
rec  = { name : 'Dave', id : 24 }
send(dest=newdest, protocol='db', idrec=rec)
```

As mentioned previously, attribute values can be any Python type that can be pickled. This includes most Python types including most immutable types, lists, dictionaries, and user-defined objects.

The SFM `recv()` function is similar to the `send()` function in that it takes a variable number of attribute-value pairs as arguments. However, the attribute-value pairs constrain which message should be received. Each received message is guaranteed to have at least the attribute-value pairs specified in the `recv()` call. The return value is an SFM message object whose attributes can be accessed using the member variable (dot) notation. The default `recv()` function is blocking; thus, if no messages match the specified criteria, the caller is

blocked until a matching message arrives. The syntax of `recv()` is:

```
message = recv([, <attribute>=<value> ] ...)
```

In its simplest form, receiving without any arguments will simply return the first message in the queue in first-in, first-out order.

```
msg = recv()
```

A more common scenario is to receive a message from a particular source. Note that the `src` attribute will appear in all messages, even though it is not explicitly specified by the sender.

```
m = recv(src='A')
```

Matching can be done on any number of attributes but the following conditions have to be satisfied: all attributes must appear in the message and their values must be equal to the values specified in the `recv()` call. Note that this does not mean the message must only have the specified attributes, just that the specified attributes must be present in the message. Here we ask for messages that came from process `A` and have the `protocol` attribute set to the value `input`:

```
m = recv(src='A', protocol='input')
```

Since the attribute names are completely arbitrary, matching can be done on any attribute. If a message does not have one or more attributes specified in `recv()`, then the message will not be accepted (it will remain in the queue to be matched later). As shown in Example 1, accessing the attribute values of a message is identical to accessing member variables of Python objects.

Example 1 Simple SFM Example

```
1 # Process A
2 mylist = range(10)
3 send(dest='B', mytag=42, input=mylist)

1 # Process B
2 m = recv(mytag=42)
3 print m
4 print m.src
5 print m.input
6 print m.mytag

# Output
Message: {'dest': 'B', 'src': 'A', 'mytag': 42, \
        'input': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}
A
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
42
```

Additionally, an attribute passed to `recv()` may be paired with a lambda expression or a named function rather than a value. In this case, no direct comparison is done but instead that function is invoked with the attribute value (from the message) as the only argument if the attribute exists in the message. If the function returns `True`, the attribute is considered to match. If all other attributes match, the message is returned. This allows the user to perform more sophisticated matching such as simulating a logical OR expression, or string matching. For example, the following code will match a message if it contains a 'tag' attribute equal to one of the two specified values:

```
m = recv(tag=lambda x: x == 23 or x == 42)
```

As mentioned above, several variants of the `recv()` function exist. All of them take a variable number of attribute-value pairs as arguments but vary in their blocking effect and return value. Table 1 summarizes them.

Name	Description
recv	Blocking receive (most commonly used).
recv_nb	Non-blocking receive. Raises an exception if no matches are found.
peek	Non-blocking peek. Returns `True` or `False` based on match result.
regcb	Registers a callback function to be invoked when a message matches the specified attributes.

Table 1: Various Receive Functions

The difference between the default `recv()` and `recv_nb()` functions is that `recv()` blocks on a condition variable when no message matching the specified attributes has been found, whereas `recv_nb()` raises an exception instead of blocking. The `peek()` function works in exactly the same way but returns `True` or `False`, indicating whether a match has been found without modifying the queue. Finally, an alternate method of receiving a message is registering a user-defined callback function with the queue. This is done by specifying the function as well as the attribute-value pairs to match. In this case when a message matching the specified attributes arrives, the user-defined function will be invoked with the message as its only argument. The user needs to register a callback function only once but it will be called for every matching message until unregistered.

3 Examples

This section presents some real examples of SFM usage. The code presented here runs in the River frame-

work [12]. As such, the code snippets use the River process naming scheme in which processes are named by universally unique identifiers, or UUIDs [8]. Also, the `send()` and `recv()` functions are provided by a virtual resource super class. A virtual resource is the River notion of a process.

Example 2 illustrates a simple way of implementing parallel dot-product computation. This excerpt is taken from the River version of parallel conjugate gradient (CG) solver. It includes a very naive implementation of the MPI `Allreduce()` function. First, the local dot product is calculated on line 2, then all processes send their local result to the root process (called parent in River terminology) on line 5. The parent collects the local results from all the processes on line 11 and sums them up, then sends the global sum back to each process on line 15. Finally, all the participating processes receive and return the global result on line 6.

Example 2 Parallel Dot-Product

```
1   def pdot(self, x, y):
2       result = dot(x, y)
3
4       if self.uuid != self.parent:
5           self.send(dest=self.parent, result=result,
6                     tag='pdot')
7           m = self.recv(src=self.parent, tag='allreduce')
8           return m.result
9
10       else:
11          for w in self.workers:
12              m = self.recv(tag='pdot')
13              result += m.result
14
15          for w in self.workers:
16              self.send(dest=w, result=result,
17                        tag='allreduce')
18
19      return local
```

Example 3 shows how SFM is used to implement flow control in our River implementation of MPI. This code fragment allows large messages to be sent in chunks so that a complete copy of a data buffer is not needed. The SFM notation makes it relatively easy to implement a chunk protocol and ensure that data is transferred in the proper order.

All the major MPI peer-to-peer and collective functions are implemented using `mpi_send_buf()` and `mpi_recv_buf()`. They allow for the sending of small messages in an eager fashion or large messages in a sequence of chunks. The parameters include: `buf` (a list of simple values), `offset` (index into `buf`), `count` (the number of values to send), `datatype` (the data type of the values in `buf`), `src` (the rank of the sender), `dst` (the rank of the receiver), `tag` (an MPI idiom for naming transfers), and `comm` (the communication namespace).

Example 3 Flow control in MPI

```
def mpi_send_buf(self, buf, offset, count, datatype,
  dest, tag, comm):
    destUuid = comm.rankToUuid(dest)
    chunksize = self.mpi_getchunksize(datatype, count)
    sendsize = count
    if sendsize <= chunksize:  # send small buf
        self.send(dest=destUuid, tag=tag,
          buf=buf[offset:offset+count], start=0, end=count,
          size=count, mtype='small', last=True)

    else:                      # send large buf
        last = False; start = offset; end = start + chunksize
        while True:
            self.send(dest=destUuid, tag=tag,
              buf=buf[start:end], start=start-offset,
              end=end-offset, size=chunksize, mtype='large',
              last=last)
            m = self.recv(src=destUuid, tag=tag, ack=True)
            if last: break
            start += chunksize
            end = start + chunksize
            if (start + chunksize) >= (offset + count):
                end = offset + count
                chunksize = end - start
                last = True

def mpi_recv_buf(self, buf, offset, count, datatype,
  src, tag, comm):
    srcUuid = comm.rankToUuid(src)
    while True:
        m = self.recv(src=srcUuid, tag=tag)
        if m.mtype == 'large':
            self.send(dest=srcUuid, tag=tag, ack=True)
        buf[m.start+offset:m.end+offset] = m.buf[0:m.size]
        if m.last: break
```

The small message code sends the entire `buf` in a single message. By setting up the proper values for `start`, `end`, and `size`, as well as setting `last` to `True`, the receiver is given all the information needed to process the send request. The large message code walks through the `buf` in `chunksize` increments. On each iteration, the values for `start`, `end`, and `size` are updated appropriately. Once the last chunk in identified, `last` is set to `True`.

In both examples given above, the data to be sent is completely specified at the `send()` call sites. This encourages rapid development because the programmer need only modify the attribute-value pairs in the `send()` parameter list and message attribute access at the `recv()` call site to add or change the data to be transmitted. Furthermore, the programmer need not worry about the structure of the data because any type can be sent as a value. Thus, SFM provides an extremely concise way to send arbitrary structured data.

4 Implementation

The SFM model consists of two main modules: the packet module that provides the basic encoding and de-

coding functionality and the queue module that provides packet matching at the receiving side. We call an SFM message a *Super Flexible Packet* (SFP). This section details sending, receiving, and queue matching.

4.1 Sending

Sending consists of serializing the specified attribute-value pairs and sending the serialized packet over the network (e.g., via a socket connection.) An SFP is created by passing a list of named parameters to the send function, which first encodes them into a binary format suitable for transmission over the network. Since named parameters in Python appear as a dictionary, we accomplish this by serializing the dictionary (or pickling, as it is called in Python) and adding minimal header information. The SFP header consists of two fields: a magic signature string and the length of the serialized payload. As mentioned earlier, an attribute is a string and the value can be of any serializable type. At least one attribute, the name of the destination process, is required. Additionally, another attribute, the name of the source process, is added to the dictionary automatically. An exception is raised if no destination is specified or if the user provides his own source. Once the data is encoded, it can be sent over a socket connection.

The SFM model does not include any low-level network communication routines, instead such communication is provided by the distributed environment. In the case of our River framework we have designed a sophisticated socket connection management scheme that includes a connection pool and support for multiplexing several logical connections (between processes) over a single physical socket connection. This makes no difference to the SFM model, since it only deals with representing the data, rather than its transportation.

4.2 Receiving

Receiving a message is a more involved process that encompasses: (1) reading data from the network; (2) decoding the data into SFPs; (3) adding SFPs to the queue of the appropriate process; and (4) performing matching to return an SFP back to the caller.

Unlike sending, there is no explicit low-level receive function available to users that would read data from a socket and return SFP objects in one fell swoop. Again, this is because the SFM model has no provisions for low-level data transfer details but instead focuses on structured representation. In the case of the River framework there is a separate network input thread responsible for centrally receiving all data from socket connections. The data is then decoded into SFPs and placed into the message queue of the appropriate process, as specified by the destination attribute within the packet. Thus, the receive functions available to application programmers operate by retrieving packets from the queue rather than blocking on I/O.

The decode function operates as follows. First, the header is decoded and parsed to determine whether a packet has been received in its entirety and to ensure that it is a valid SFP. If the packet is incomplete, it is not processed until more data is available to complete the remaining part. Otherwise, the payload is deserialized into a Python dictionary and then used to construct a packet object that represents the SFP. The packet class overrides the internal `__getattr__` method, allowing users to access the packet's data by referencing attribute names using member field (dot) notation. This lets the user refer to the attributes contained within the packet using the same names as originally specified by the sender.

Two things are returned: a list of successfully unserialized packet objects (in case the input buffer contained more than one), which could be empty if there was not enough data to complete even one packet, and any leftover data, which could be an empty buffer, or the entire input buffer. Since the decode function does not preserve any state, the leftover data should be saved by the caller and then prepended to the next input data obtained from the network before the next round of decoding is attempted. If the decode function determines that the input buffer does not contain a valid serialized SFP, an exception is raised. This should never be the case when a reliable low-level communication protocol, such as TCP, is employed.

Once we have a list of SFP objects, we determine the destination process to which each packet was sent by using the `dest` attribute within. The packet is then added to the message queue of that process. All these steps happen within the context of the network input thread, which lastly performs a notify on a condition variable attached to the message queue, in case a process was blocked on a `recv()`.

4.3 Queue Matching

The message queue consists of an ordered list of received SFPs (in a first-in, first-out order) and a condition variable, which is used to block a process when no packet can be returned. In its basic form the receive function (the queue get method), like the send function, takes a variable number of attribute-value pairs (again, as Python named parameters) but uses them as a key to

match against packets in the queue. That is, for a packet to match, all attribute-value pairs passed to the receive function must appear in the packet and have the exact same value. To perform matching we linearly walk the queue of packets and check the specified attributes against every packet. If an attribute was specified to the receive function but does not appear in a packet or its value in a packet is different than the one specified, matching fails and we proceed to the next packet. The basic receive function is blocking and if no packets are present in the queue or none of them match, the calling process is put to sleep on a condition variable. Once a new packet is added to the queue, the network input thread will signal on the condition variable, waking up the sleeping process and allowing it to retry matching.

If the value of an attribute is a lambda expression or a function, rather than a simple data type, no direct comparison is done but instead that function is invoked with the attribute value (from the packet) as the only argument. If the function returns `True`, the attribute is considered to match. If all other attributes match, the packet is returned.

Additionally, we provide other variants of receive, such as a non-blocking receive that raises an exception if no packets were matched and a peek function that returns a boolean value indicating whether a matching packet can be retrieved from the queue. The matching semantics are exactly the same as the basic receive: any number of attribute-value pairs to perform the matching on are passed in as input parameters. Moreover, an application can register a callback function, which will be invoked with the packet as a single argument if the latter matched the specified attributes. In River, such callbacks can be invoked asynchronously by the network thread.

5 Results

To determine the cost of the SFM mechanism we constructed a simple ping-pong benchmark. Using this benchmark we compare performance of different types of mechanisms that could be used to send data between Python processes. For the experimental platform we used two identical machines with the following configuration: dual Opteron 270 processors, 4 gigabytes of RAM, and a Gigabit NIC. Both were connected to an isolated Gigabit network.

Table 2 shows the latency and bandwidth of four different implementations of a ping-pong benchmark that transmits a character array of a given size back and forth between two processes. The first version, `sockets`, simply sends the array from sender to receiver and back using the Python interface to the socket library; no encoding of the data is done. The `pickle` version is an example of how a programmer could use the `pickle` module in Python to transfer structured data to a remote machine. It demonstrates the cost of pickling a Python object and sending it over the network. Because we do not know the size of pickled data a priori, we prepend a small header that contains this information to the data. This incurs a performance penalty due to an additional copy before sending. The `SFM` version uses `encode()` and `decode()` functions with the named argument notation provided by the SFM model to serialize the data. Finally, `SFM w/Q` implementation adds trivial use of the queue mechanism provided by the SFM model by putting the received packet to the queue and then retrieving it. Because this involves operations on a condition variable within the queue, it results in an additional performance loss.

	128 bytes		16 kb		64 kb	
mechanism	lat	bw	lat	bw	lat	bw
sockets	48	20	220	568	642	778
pickle	73	16	275	453	964	518
SFM	84	14	285	439	979	511
SFM w/Q	97	13	300	417	984	508

Table 2: Ping-Pong Latency (lat, microseconds) and Bandwidth (bw, Mbps)

Both `SFM` versions introduce overhead that results in lower bandwidth and higher latencies but significantly simplify the code that a programmer would normally have to write to send structured data across the network. As discussed earlier, the SFM queue also allows powerful packet matching based on attributes and their values. These results represent the worst-case scenario since a typical distributed program would consist of both communication and computation. Additionally, our queue implementation has not been optimized yet. These preliminary results indicate that any additional Python processing in the critical path of network communication results in significant overhead. For instance, both encoding and decoding functions incur overhead due to copying of buffer data. Also, Python libraries do not provide any scatter/gather I/O operations, which would be beneficial in this case.

6 Experience

Super Flexible Messaging has been successfully used in our implementation of the River framework [12], a par-

allel and distributed programming environment written in Python that targets conventional applications. The River core interface is based on a few fundamental concepts that enable the execution of code on multiple machines and provide a flexible mechanism for communication among them.

The main features of River include a simple programming model, easy program deployment, and reliable execution. The goal is to allow a user to leverage multiple machines, i.e., desktops and laptops, to improve the performance of everyday computing tasks. We provide an easy way for the application programmer to exploit the low-hanging fruit of parallelism in their tasks without requiring in-depth knowledge of parallel computing or familiarity with complicated parallel programming interfaces.

We employ SFM for basic communication between processes, both at the system and application levels. All the internal communication within River, such as machine discovery, allocation, and process deployment is achieved through the use of SFM. Any possible conflicts between system and application packets are avoided by giving application processes different names from the system processes. (In River the processes are named with universally unique identifiers, or UUIDs.) The flexibility and ease of use of this model combined with powerful and elegant features of the Python programming language have made the rapid prototyping of our system extremely efficient and allowed us to explore many different approaches for which we otherwise would not have time. For instance, the flexibility afforded by the SFM model has allowed us to quickly implement extensions to the River framework, such as an implementation of the MPI library, remote access methods, a new task farming language called Trickle [3], and a MapReduce extension [6].

7 Conclusions

Super Flexible Messaging allows developers to focus on program logic and not on cumbersome data transfer details. With SFM there is no need to define a packet structure or a protocol format and the data being exchanged consists of simple attribute-value pairs, where an attribute is a string, and the value can be of any serializable type. The Python implementation of SFM leverages off the Python concept of named parameters, providing a straightforward, yet very powerful syntax for sending and receiving data. SFM can be implemented in other dynamically typed languages that support named parameters (or simulated named parameters) in function invocation such as Ruby [13] and Lua [9].

Acknowledgments

We thank members of the River research group for their feedback and technical support: Yiting Wu, Brian Hardie, Tony Ngo, Jennifer Reyes, and Joe Gutierrez. We also thank Peter Pacheco for reviewing an earlier version of this paper. This work was partially supported by a University of San Francisco faculty development research grant.

References

[1] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[2] David M. Beazley. *Python essential reference*. Sams Publishing, third edition, 2006.

[3] G. D. Benson and A. S. Fedosov. Python-based distributed programming with trickle. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, June 2007. CSREA Press.

[4] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer, first edition, 2005.

[5] S. G. Caglar, G. D. Benson, Q. Huang, and C. Chu. USFMPI: A multi-threaded implementation of MPI for linux clusters. In *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, Marina del Rey, CA, December 2003.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of Operating Systems Design and Implementation*, pages 137–150. USENIX, 2004.

[7] Cay S. Horstmann and Gary Cornell. *Core Java 2, Volume II: Advanced Features*. Prentice-Hall PTR, seventh edition, 2004.

[8] Paul J. Leach, Michael Mealling, and Richard Salz. A universally unique IDentifier (UUID) URN namespace. Internet proposed standard RFC 4122, July 2005.

[9] The programming language Lua. http://www.lua.org.

[10] Peter S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers, 1997.

[11] The Python programming language. http://www.python.org.

[12] River. http://www.cs.usfca.edu/river.

[13] The Ruby programming language. http://www.ruby-lang.org.

[14] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX network programming: The Sockets Networking API*, volume 1. Prentice-Hall PTR, third edition, 2004.