TCP over IP over UDP

cs168 Computer Networks
Fall, 2014

Da Yu & Yiming Li
Department of Computer Science
Brown University

Contents

1 About

One monitor, two men(Dev and QA by turns), seven weeks for peer coding, more than 70 scala files, more than 100 commits, more than 4400 lines code, sending 5GB data reliably in 11MB/s (Effective Bandwidth). Now, we have own TCP over own IP over UDP in Scala.

2 Introduction

Following RFC 793, we have implemented majority of state machine, such as three-way shake, teardown, adhere to the flow control window, retransmit in building connection, teardown and loss data, fast retransmit and congestion control. Most of corner cases are considered in the state machine, which focus on the shake and teardown. Based on testing in Mac, our Scala TCP/IP can achieve effective bandwidth more than 10MB/s (80Mb/s) on two nodes connected directly to each other. If any loss occurs (about 5%) without congection control, we still reach 8MB/s (64Mb/s). Congection control brings the speed little slow if there is loss.

3 Architecture

Application Layer

There are two kinds of users: server or client. As a server (passive open), it turns on listening and wait for accepting the remote clients. Once connection, the server should assign one new socket with Transmission Control Block (TCB). Note, this socket doesn't be binded to one new port and only use same port of this listening server. As a client, it can connect (active open) to remote server and send some data.

TCP Layer

TCP controls all the ports and socket numbers. Any segment delivered from IP layer is sent to demultiplexing buffer. TCP will assign to each TCB. Any segment that needs to send out is pushed into multiplexing buffer and wait for giving IP layer. For each TCB, it records the seq and ack number in order to help retranmit and avoid duplicate data.

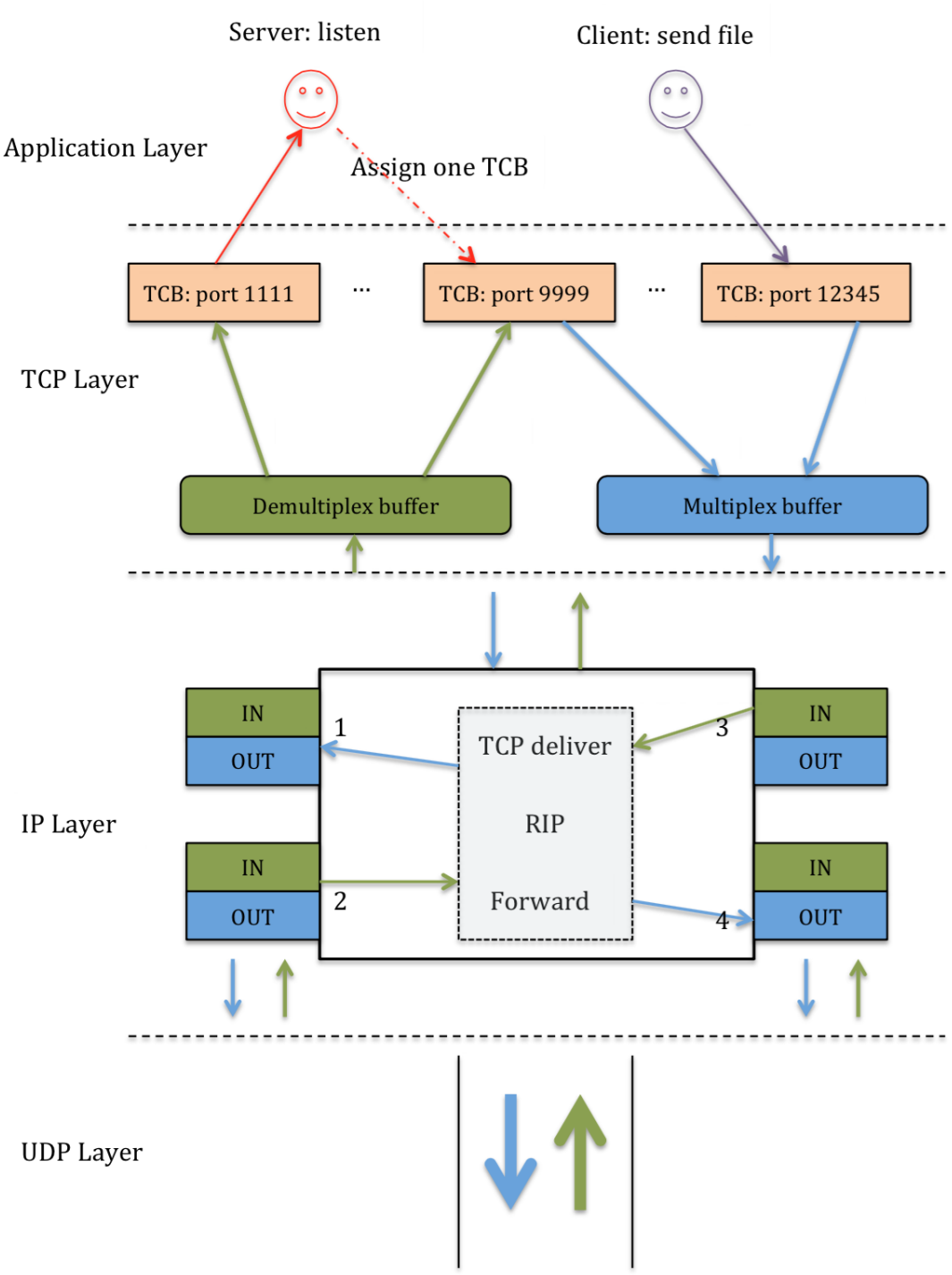

Figure 1: Architecture of TCP/IP node

IP Layer

We simulate several network interface controllers (NIC) for node. Each interface has own input buffer and output buffer as a router. The thread runs round robin to capture packet from each interface or output to each interface. In main controller, we have three handlers. Forwarding handler is to continue to passing the packet to next router. RIP handler is to update the routing table depending algorithm. Finally, TCP handler only remove IP head and pushes the payload (segment) to TCP layer.

UDP Layer

Only call Scala API to send out packet or receive packet. Note, if we send quickly, UDP layer will fill buffer and drop several packets depending implement in the low level.

4 IP Design

We treat UDP as our link layer, since the network model of UDP and Ethernet are similar. Our own IP and TCP are built on top of real UDP.

4.1 NodeInterface and LinkInterface

NodeInterface: Abstraction of Node, contains several LinkInterfaces belonged to this node. Also, it implements logics for forwarding, routing, multiplexing/demultiplexing to up-layer TCP, etc. LinkInterface: Abstraction of NIC, contains information about the NIC, includes NIC information, buffer status, etc. Buffers in LinkInterface emulates queues in real NICs, storing packets that need to send to the network or receive from the network.

4.2 Threads

- User input thread (main)
- Sending thread: Fetch packets from each output Queue(Buffer) in LinkInterface and push these packets into the network

- Receiving thread: Get packets from network, and store these packets to corresponding input Queue(Buffer) in LinkInterface, based on virtual IP
- HandlerManager thread: Get packets from input Queue and send to corresponding handler in the Node, based on protocol in the IP head (0 for RIP, 200 for Data, 6 for TCP)
- PeriodicUpdate (5s) thread: Periodicly generate hearbeat RIP advertisements to all neighbors, includes all routing table and all up interfaces
- Expire (12s) thread: Remove expired routes in the routing table, based on periodicUpdate received by neighbors

4.3 Queues(Buffers) in LinkInterface

Input Queue (inBuffer): Store packets received by the Receiving thread from real UDP, and wait for HandlerManager thread to retrieve and get to the corresponding handler. Output Queue (outBuffer): When handlers finish corresponding logic, they may modify the original packets or generate new packets. After that, these packets will be stored in correct output queues, waiting for sending thread to send to the real network

4.4 Lock

- inBuffer/outBuffer - mutex for read/write
- routing table - read/write lock
- expire - read/write lock

4.5 RIP

The node will generate three kinds of RIP advertisement:

- Periodic Updates: Send all routing table entries and interfaces
- RIP updates: Only send modified updates

- RIP response: Send all routing table entries and interfaces to reply a RIP request

4.6 Convert Number/Object

We follow OOP model to design and implement the whole TCP/IP. So, in the node, everything is object, we convert all packet(bytes) from the network to object when we need to handle logic operations. Meanwhile, when we need to push it back to the network, we convert these objects to bytes, based on IP head. In addition, in Scala, we also need to do bit shifting ourself, since Scala doesn't support uint16.

5 TCP Design

5.1 Build TCP on top of IP

We register TCPHandler to IP, which means when the node receives a IP packet(sent to this node) with protocol number 6, TCPHandler will send this packet to the TCP module. Multiplexing and Demultiplexing glue TCP and IP together. Both of them are implemented by queue:

- Multiplexing: Store packets generated by up layer TCP connections, which will be sent to remote nodes later
- Demultiplexing: Store packets (contains TCP segment inside) received from remote nodes and check the TCP checksum. Find the corresponding connection (transmission control block - TCB) and call receive function for that TCB. If the packet fail to pass the checksum, drop the packet. Or, send packet with reset bit when the port number is illegal.

5.2 TCP Connection(TCB)

- Basic information: src IP, dst IP, src port, dst port (listen server is special).
- State machine: we have implemented most TCP state machine. There are two ways to change state, one is to call TCP API. Another one is to receive specific TCP segment. We will drop illegal TCP segments.

- Sending buffer: We implement sending buffer by two arrays - writing buffer and flight buffer. Writing buffer stores data that need to be sent whereas flight buffer stores data that has already been sent before and waits for ack. When the user needs to send data, these data may be appended into the tail of the writing buffer. When the sending data thread is ready to send data, the thread will pop data from the head of the writing buffer and append into flight buffer's tail. When acks arrives, the first ack-seq bytes will be removed from the flight buffer. Note that, we blocked the buffer if the space is not enough to write any new data.

- Receiving buffer: We also implement receiving buffer by two arrays - receiving buffer and pending buffer. Pending buffer stores data that is carried by valid TCP segments (seq in the sliding window). Actually, pending buffer stores a tuple - ([Begin index, end index), data), which may be pretty easy for us to assemble received data. For example, if we need to send data [a, b, c, d], the receiver may receive three segments in arbitrary order: [d], [b, c], [a, b]. The pending buffer will store as follows:

  - before: empty, add: (3, 4, [d]) ⇒ after: (3, 4, [d]), needs to wait data begin at 0
  - before: (3, 4, [d]), add: (1, 3, [b, c]) ⇒ after: (1, 4, [b, c, d]), needs to wait data begin at 0
  - before: (1, 4, [b, c, d]), add: (0, 2, [a, b]) ⇒ after: (0, 4, [a, b, c, d]), deliver to receiving buffer

  Here, we can see if there is a hole in the pending array, it will store them or merge them until the data starts from 0 and deliver continuous to the receiving buffer, which can be read by the up layer applications. Note that, we maintained the offset for seq and ack numbers to check whether the TCP segment is valid.

- Sequence/acknowledge number: Sequence number will be generated randomly when the client initiates a connection. Also, during the three-way handshake, these two TCP peers will synchronize ack numbers.

- Flow control: The sender can know the advertised window size (available buffer size) of receiver by checking the window size field in the TCP head. When this field becomes 0, the sender will keep sending 1-byte segments to probe the remote window size.

- Congestion control: We implemented TCP Reno. When the sender receives 3 dup-acks, it may cut the congestion window in half. Meanwhile, while timeout happens, it may set the congestion window to 1 MSS.

- Three duplicate acks: The TCB always records ack for each receiving segment. When receiving three duplicate acks, we cut congestion window in half and read one MSS data from flight buffer to do fast retransmit.

- Timeout:

  - Establishing or teardowning a connection timeout: it will do 3 retransmit SYN or FIN for establishing or teardowning a new connection. Once it receives a valid ack back, it will cancel the timeout. Otherwise, after 9 seconds (3 * 3s), it will change the state to CLOSE and remove this connection.
  - Data sending timeout: When the sender fails to receive valid ACK in one RTO, it will retansmit all pending data (in the sliding window) in the flight buffer and reset the timer. Meanwhile, it will set the congestion window size to 1 MSS. Once the sender timeouts 20 times, it will set this connectino to CLOSE state and disconnect.

- We calculate RTO based on RTT. When sending TCP segment, the sender records the time and waits for receiver's reply. When receiving reply corresponding to the previous sent TCP segment, the sender will compute SRTT and RTO.

  - SRTT = (ALPHA * SRTT) + ((1-ALPHA) * RTT)
  - RTO = min(1000, max(10, 2 * SRTT)) - unit: millisecond

5.3 Threads

- Demultiplexing: We implemented demultiplexing by a control thread, which contains a fixed thread pool. When a segment is fetched from the demultiplexing buffer, one thread in the thread pool will deliver this segment to the TCP level.
- Multiplexing: We implemented multiplexing by a control thread, which also contains a fixed thread pool. When a segment need sent from the multiplexing buffer, one thread in the thread pool will deliver this segment to the IP level.
- Data sending: This thread only works for sending data if and only if there is some data in sending buffer or receiving data, which is created after the three-way handshake finished.
- Connect or teardown timeout: This thread only works in the state of three-way handshake and teardown. It will set timeout thread once sending out SYN or FIN segment. Timeout will be cancelled if state changes.
- Data sending timeout: similar to connect or teardown timeout thread, this thread is created after setting established state. It will wait for some time after sending some segments. Once receiving data, the timeout will be reset. Otherwise, it will resend all the flight data under sliding window size and reset the timeout.
- Accept: When the server is listening on a port, a client try to connect to this port and the server will store this connection into a queue. The accept thread will dequeue a pending connection.
- Receive file: application level, only when starting to receive data into file.
- Sending file: application level, only when starting to send data from file.

5.4 Lock

three levels, we keep the lock order as TCP $\rightarrow$ TCP connection $\rightarrow$ Sending/receiving buffer, we avoid locking high level in the low level.

- TCP: the synchronized lock controls all sockets, mapping sockets to each connection, mapping client/server tuples to each conncection. It will make sure they are synchronized to do any modifications or read from global variables.
- TCP connection(TCB): the synchronized lock controls each connection. It will make sure sending segments and receiving segments will not be handled by the same time, because of the consistency of acks and seqs.
- Sending/receiving buffer: the synchronized lock controls each buffer, which aims to avoid reading and writing at the same time.

5.5 TCP API

- virSocket: register one socket, it will find the minimum free socket number starting from 3
- virBind: given port number, bind to the socket
- virListen: give the socket and set to LISTEN state
- virConnect: connect to a given ip and port until established or 3 re-transmitted SYNs timeout
- virAccept: poll one socket from listening queue and implement three-way handshake
- virRead: read from given socket
- virWrite: write to given socket
- virShutDown: depending on the type, block read/write/both, here, it doesn't remove from socket array
- virClose: it will send FIN to remote connection and remove itself from socket array, blocking writing

5.6 Exception(Scala)

- BoundedSocketException: the socket has been used
- DestinationUnreadable: the remote ip is unreachable
- ErrorTCPStateException: error tcp state when expecting another state
- InvalidPortException: port is not valid. Range should be 1024 - 65535
- InvalidSocketException: the socket should from 3 to 65535, (0 - stdin, 1 - stdout 2 - stderr)
- PortUsedUpException: the port from 1024 to 65535 has been used
- ReadblockException: reading has been closed due to shutdown
- ServerCannotShutdownException: server(listen state) cannot be shutdown of read or write type, only can be shutdown by both type
- ServerHasCloseException: server has closed
- ServerClosedException: Socket has been closed
- SocketUsedUpException: the socket from 3 to 65535 has been used
- UnboundSocketException: socket is has not been bounded when using
- UninitialSocketException: socket has not been initiated when using
- UsedPortException: port has been used
- WriteBlockException: writing has been closed

6 Extra Credit

6.1 IP

IP (TCP doesn't send MSS larger than mtu):

1. The minimum mtu is set to the max head + minimum offset (60 + 8 = 68)

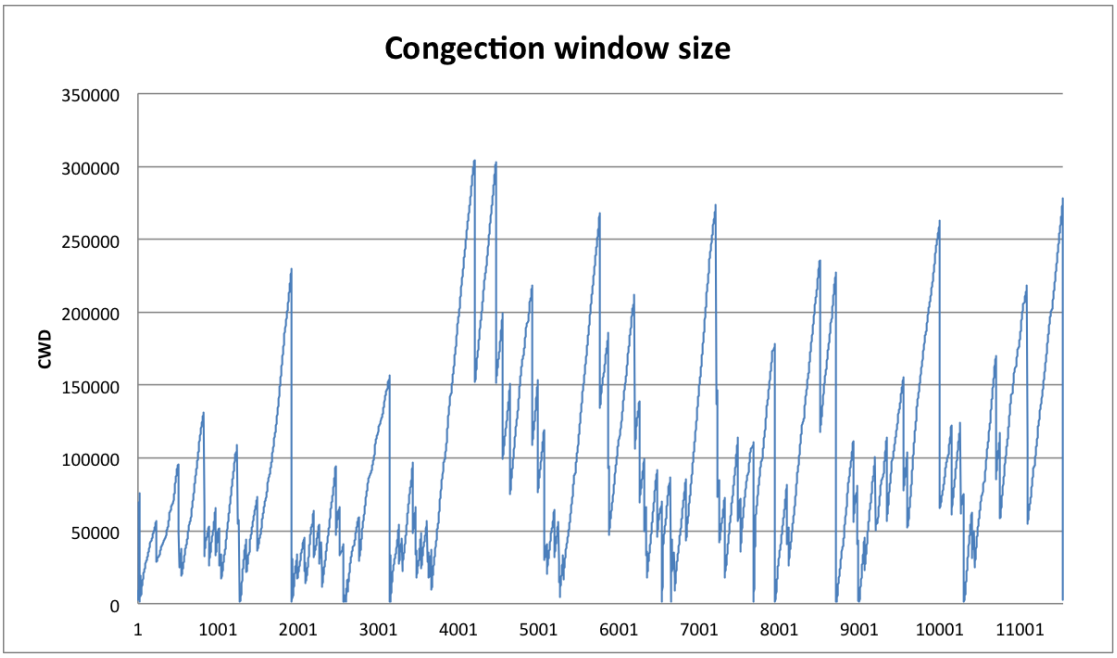

Figure 2: Congection control

2. Fragmenting is done before sending the packet, while assembling the packet is done before receiving the packet that corresponding to one of interface of that node

3. The time out (20s) of assembling is similar to that of expire

6.2 TCP

See the figure for congection control of sending 47271516 bytes file.

1. Slow Start: start from one MSS, sendLen-the length for removing from flight buffer after receiving, totalLen-the total length of flight buffer

$$cwd >= threshold : cwd = cwd + (sendLen/totalLen) * MSS$$

$$cwd < threshold : cwd = cwd + sendLen$$

2. Three duplicate acks: we set to half of congection window size, change threshold to this new congection window size

3. Data sending timeout: we set threshold to half of congection window size and set congection window size to one MSS

7 Default values

1. (De)Multiplexing buffer size: 2 * 1024 * 1024 * 1024 - 1 Bytes
2. Flow buffer size: 64 * 1024 - 1 Bytes
3. MSS: 1400 - 40 Bytes
4. MTU: 1400 Bytes
5. Listen pending queue: 64 * 1024 - 1 Bytes
6. MSL: 2 * 60 * 1000 ms
7. Retransmit times of connection or teardown: 3
8. Timeout of connection or teardown: 3 * 1000 ms
9. Retransmit of data fro the same ACKs: 20
10. Threads pool: 10
11. Socket: 3 - 65535
12. Port: 1024 - 65535
13. Loss: 0%, the range is [0%, 100%]
14. Trace for multiplex/demuliplex: false (turn off debug printing)
15. RTO: initial - 30 ms, upbound - 1000 ms, lowbound - 10 ms
16. I/O file: 1024 * 10 bytes

8 Performance

Test 1GB data, based on Mac 16 GB 1600 MHz DDR3, 2.7 GHz Intel Core i7

1. Effective bandwidth (a perfect link, two nodes, no loss): 10 MB/s (80Mb/s)
2. Effective bandwidth (a perfect link, two nodes, 2% lost): 5 MB/s (40Mb/s), once lost, it may affect congection window, timeout and set to one MSS
3. Effective bandwidth (a perfect link, two nodes, 2% lost, no congection control): 10 MB/s (80Mb/s)
4. Effective bandwidth (a perfect link, two nodes, 5% lost, no congection control): 8 MB/s (64Mb/s)

Test 5GB data for all the sequence number range from 0 to $2^{32} - 1$, it is successful and effective bandwidth: 11MB/s (88Mb/s)

9 Limitation or Bug

1. All interfaces will be sent for periodic update, which means if the user brings down one interface, it will update after 5s or remove from other nodes after 12s. We think it makes sense in this senario: if the user brings down an interface, and brings up quickly (reboot). We don't need to update the RIP for this situation.
2. All the codes are compiled to byte code, then running on the JVM. It will be a little slower than other binary code. In addition, if we send data file to remote node with more than one hops, it will be very slow. This is due to some loss in each node, which is limited by UDP buffer.

10 Appendix

Depending on VisualVM, we provide statistics for sending 400MB data.

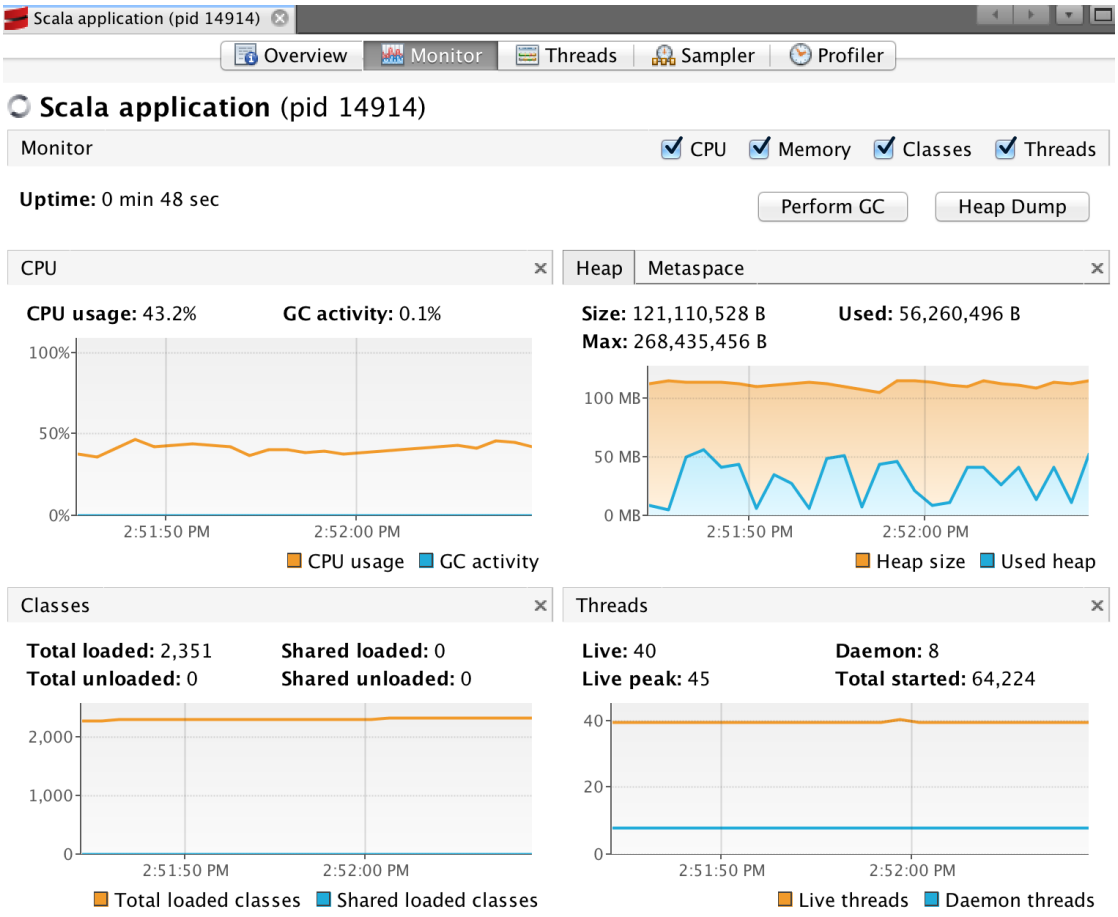

Figure 3: Monitor of Scala Application

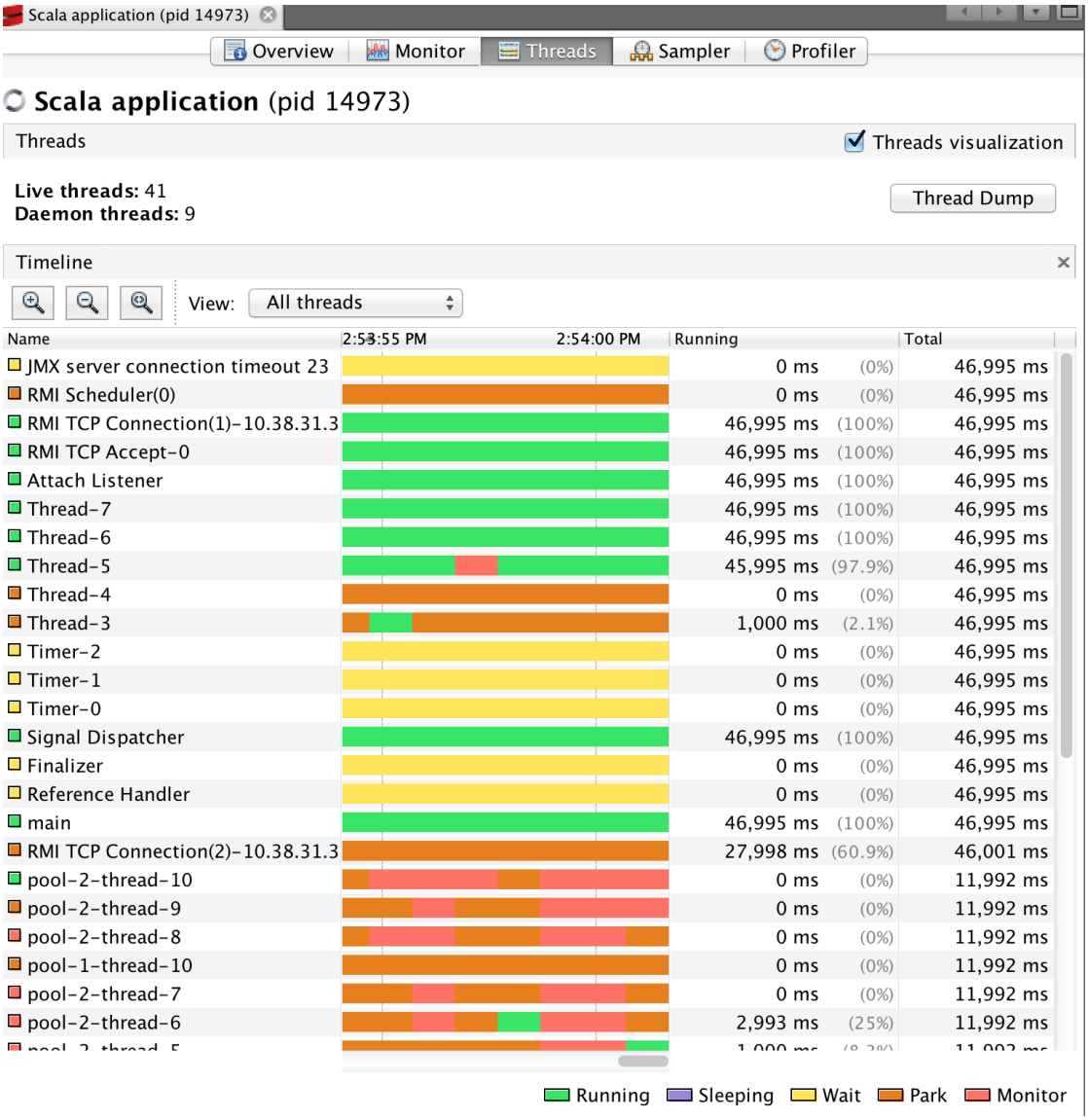

Figure 4: Threads

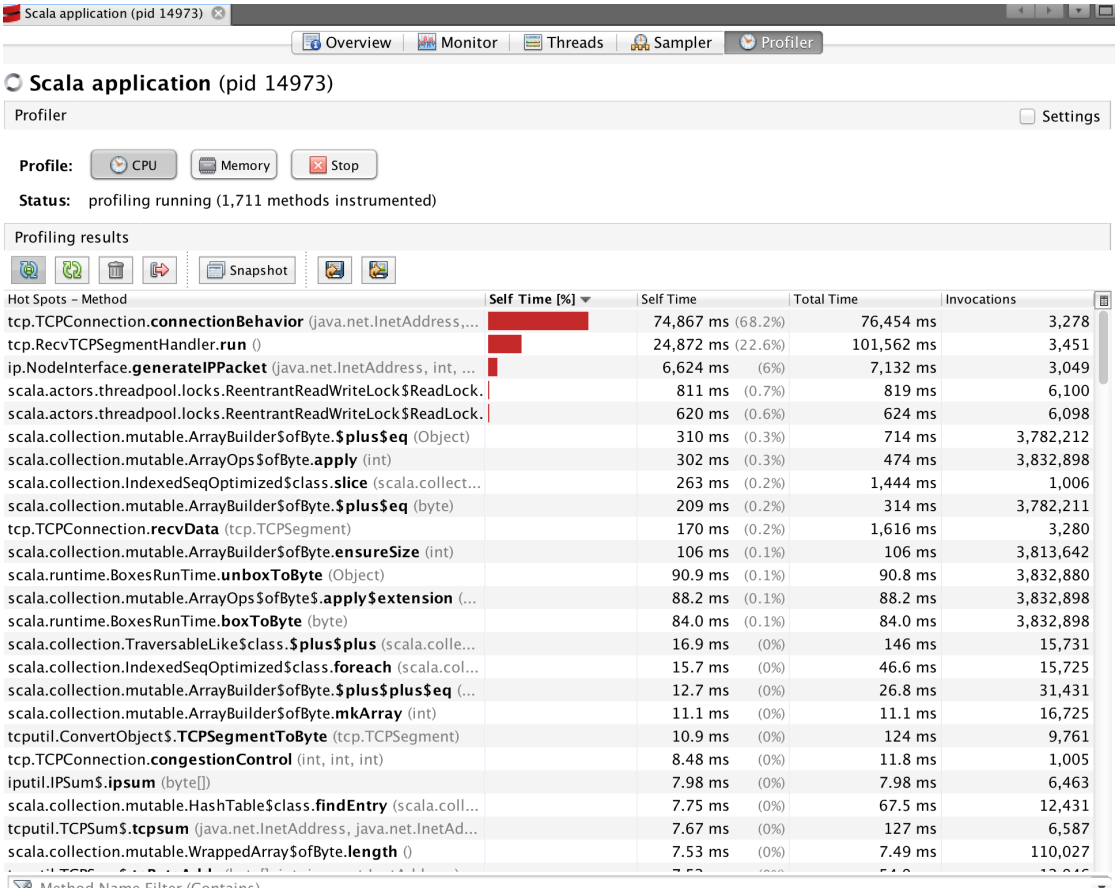

Figure 5: Cost of CPU for each function

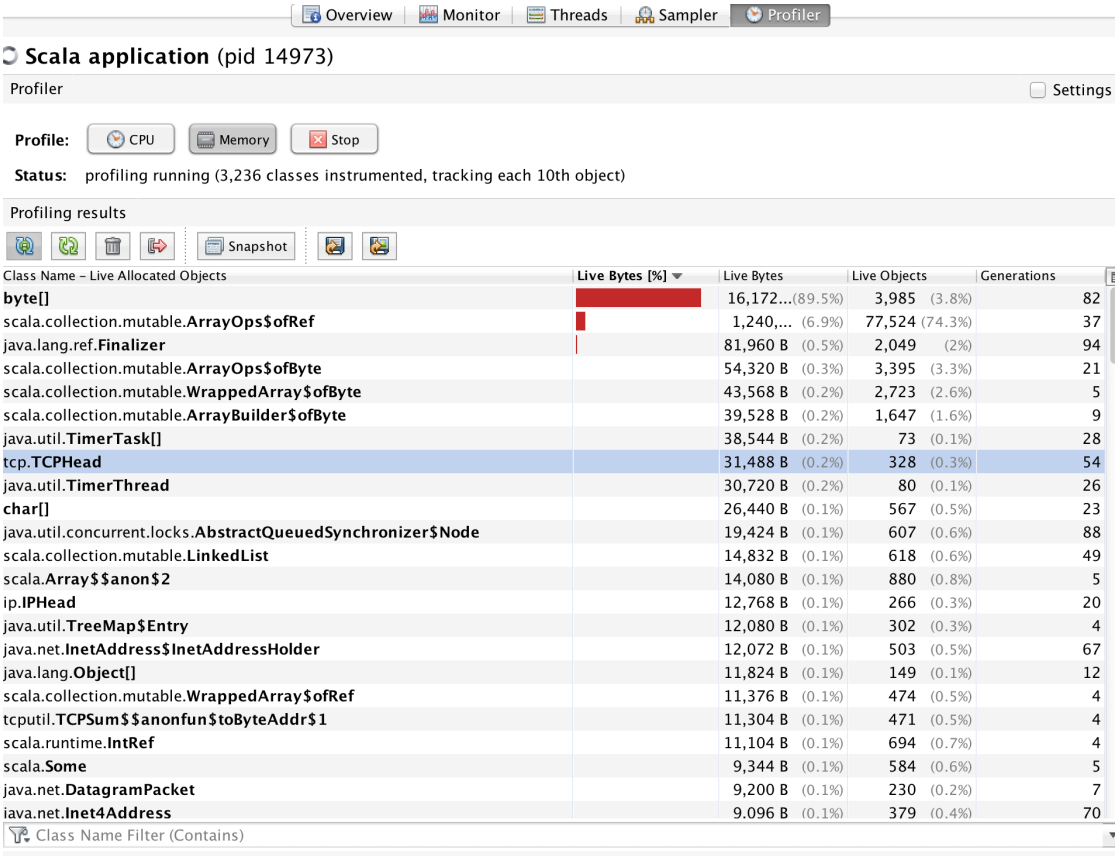

Figure 6: Cost of Memory for each function