Multicast

Point-to-point connections handle a lot of communication needs, but passing the same information between many peers becomes challenging as the number of direct connections grows. Sending messages separately to each recipient consumes additional processing time and bandwidth, which can be a problem for applications such as streaming video or audio. Using multicast to deliver messages to more than one endpoint at a time achieves better efficiency because the network infrastructure ensures that the packets are delivered to all recipients.

Multicast messages are always sent using UDP, since TCP requires an end-to-end communication channel. The addresses for multicast, called multicast groups, are a subset of regular IPv4 address range (224.0.0.0 through 230.255.255.255) reserved for multicast traffic. These addresses are treated specially by network routers and switches, so messages sent to the group can be distributed over the Internet to all recipients that have joined the group.

Note

Some managed switches and routers have multicast traffic disabled by default. If you have trouble with the example programs, check your network hardware settings.

Sending Multicast Messages

This modified echo client will send a message to a multicast group, then report all of the responses it receives. Since it has no way of knowing how many responses to expect, it uses a timeout value on the socket to avoid blocking indefinitely waiting for an answer.

```
import socket

import struct

import sys


message = 'very important data'

multicast_group = ('224.3.29.71', 10000)
```

```
# Create the datagram socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)


# Set a timeout so the socket does not block indefinitely when trying
# to receive data.
sock.settimeout(0.2)
```

The socket also needs to be configured with a time-to-live value (TTL) for the messages. The TTL controls how many networks will receive the packet. Set the TTL with the IP_MULTICAST_TTL option and setsockopt(). The default, 1, means that the packets are not forwarded by the router beyond the current network segment. The value can range up to 255, and should be packed into a single byte.

```
# Set the time-to-live for messages to 1 so they do not go past the
# local network segment.
ttl = struct.pack('b', 1)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ttl)
```

The rest of the sender looks like the UDP echo client, except that it expects multiple responses so uses a loop to call recvfrom() until it times out.

```
try:

    # Send data to the multicast group
    print >>sys.stderr, 'sending "%s"' % message
    sent = sock.sendto(message, multicast_group)
```

```
    # Look for responses from all recipients
    while True:
        print >>sys.stderr, 'waiting to receive'
        try:
            data, server = sock.recvfrom(16)
        except socket.timeout:
            print >>sys.stderr, 'timed out, no more responses'
            break
        else:
            print >>sys.stderr, 'received "%s" from %s' % (data, server)

finally:
    print >>sys.stderr, 'closing socket'
    sock.close()
```

Receiving Multicast Messages

The first step to establishing a multicast receiver is to create the UDP socket.

```
import socket
import struct
import sys

multicast_group = '224.3.29.71'
server_address = ('', 10000)
```

```
# Create the socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)


# Bind to the server address
sock.bind(server_address)
```

After the regular socket is created and bound to a port, it can be added to the multicast group by using setsockopt() to change theIP_ADD_MEMBERSHIP option. The option value is the 8-byte packed representation of the multicast group address followed by the network interface on which the server should listen for the traffic, identified by its IP address. In this case, the receiver listens on all interfaces using INADDR_ANY.

```
# Tell the operating system to add the socket to the multicast group
# on all interfaces.
group = socket.inet_aton(multicast_group)
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
```

The main loop for the receiver is just like the regular UDP echo server.

```
# Receive/respond loop
while True:
    print >>sys.stderr, '\nwaiting to receive message'
    data, address = sock.recvfrom(1024)


    print >>sys.stderr, 'received %s bytes from %s' % (len(data), address)
    print >>sys.stderr, data
```

```
    print >>sys.stderr, 'sending acknowledgement to', address
    sock.sendto('ack', address)
```

Example Output

This example shows the multicast receiver running on two different hosts, A has address 192.168.1.17 and B has address 192.168.1.8.

```
[A]$ python ./socket_multicast_receiver.py

waiting to receive message
received 19 bytes from ('192.168.1.17', 51382)
very important data
sending acknowledgement to ('192.168.1.17', 51382)

[B]$ python ./socket_multicast_receiver.py

binding to ('', 10000)

waiting to receive message
received 19 bytes from ('192.168.1.17', 51382)
very important data
sending acknowledgement to ('192.168.1.17', 51382)
```

The sender is running on host A.

```
$ python ./socket_multicast_sender.py
```

sending "very important data"

waiting to receive

received "ack" from ('192.168.1.17', 10000)

waiting to receive

received "ack" from ('192.168.1.8', 10000)

waiting to receive

timed out, no more responses

closing socket

The message is sent one time, and two acknowledgements of the outgoing message are received, one from each of host A and B.

Source: http://bip.weizmann.ac.il/course/python/PyMOTW/PyMOTW/docs/socket/multicast.html