A Distributed Publish/Subscribe Broker

Due at 11:59pm on Monday, 28 November 2022

One of the interprocess communication mechanisms that we briefly discussed in Lecture04 was a publish/subscribe (PS) system. In a PS system, a server (the broker) maintains a set of named channels (sometimes called mailboxes). A client program (a subscriber) can subscribe to a channel, and any messages sent to that channel subsequent to its subscription are then sent to that program. Other client programs (termed publishers) are able to post messages to a channel; such messages are then sent on to all subscribers at the time the messages are posted.

This basic structure is quite simple. The difficulty is that subscribers often disappear without informing the broker. The broker must constantly be managing the set of subscribers associated with a channel during its operation.

You will be building a Distributed Publish/Subscribe Broker (PSB); each version provides an increasingly complex set of capabilities.

There are 4 parts to this project, the first 3 of which are mandatory, with the 4th being for extra credit. The objectives of the project are to give you experience writing a distributed server that meets a specification, practice using different data structures to construct a significant application, and familiarity with different system programming notions in Linux and C.

All coding must be done in the C programming language; it must be compilable and runnable in the Linux virtual machine environment. Unlike Project 1, you may use any library functions available from section 3 of the manual on your Debian virtual machine.

You should tackle this problem in four steps, outlined below. Each step should be in a separate source file, psbv?.c, where ? is replaced by 1, 2, 3, or 4. The solution at each step is graded separately.

The usage string for all versions of the broker is:

usage: ./psbv? [-f <filename>]

where ? is replaced by 1, 2, 3, or 4.

All versions of PSB should catch INT signals; upon receipt of an INT signal, it should gracefully terminate all processing.

1 PSB echos the requests it receives

The goal of Part 1 is to develop the first version of the PSB such that it can echo back the requests it receives from the echoclient program supplied in the starting archive. PSB v1 will perform the following steps:

- Check the arguments provided; if "-f <filename>" has been specified, <filename> contains a set of channel names, one per line. You are to open this file, read each channel name, and print "Creating publish/subscribe channel: %s\n" with the "%s" replaced by the channel name read from the file. Obviously, after you have processed the entire file, you need to close it.

- Initialize the BXP runtime[1] so that it can create and accept encrypted connection requests.
- Create a thread that will receive requests from client applications.
- Respond to each such request by echoing back the received request along with a status byte; the first byte of the response is '1' for success, '0' for failure. Thus, if the request is in the character array query, the response to send back would be created by sprintf(response, "1%s", query); for this version of PSB, you will ***never*** respond with a failure indication.
- Except for initialization failure, open failure of the initialization file, and receipt of an INT signal, PSBv1 should never exit.

2 PSB parses each request and sets the status byte accordingly

Successful completion of Part 1 gives you a basic PSB for which the network plumbing works correctly. PSBv2 now extends[2] PSBv1 by validating each received request; it will still echo the received request, but it will set the status byte appropriately if it is a legal request or not.

The legal queries to the PSB are strings of the following form:

CreateChannel|<name>

> Creates a channel named <name>; if successful, returns a channel id; if <name> already exists, returns the channel id associated with it; if unsuccessful, returns "0"

DestroyChannel|<chid>

> Destroys the channel indicated by <chid>; if successful, purges all messages still to send out to subscribers, and returns <chid>; if unsuccessful, returns "0"

ListChannels

> Returns a comma,separated list of channel names in alphabetic order.

ListSubscribers|<name>

> Returns a list of subscribers to the channel named <name>, one per line. There is no particular order to the lines.

Publish|<name>|<message>

> Adds <message> to the channel for delivery to all subscribers at the time the <message> is received. If successfully added, returns "1"; if not, returns "0"

Subscribe|<name>|<clid>|<host>|<service>|<port>

> Subscribes the service at <host>, <service>, <port> to the named channel; associates <clid> with the subscription; if successful, returns <svid>; if not, return "0".

[1] See section 5 below for a discussion of the BXP runtime that you will be using.
[2] It must do everything that PSBv1 does and more.

Unsubscribe|<svid>

Cancels the subscription denoted by <svid>; returns "1" if successful, "0" if not.

Thus, the legal things you can ask the server to do are create/destroy a PS channel, list the channels known to the broker, list the subscribers known to a particular channel, publish a message on a channel, and subscribe/unsubscribe from a channel. The parameters to these requests have the following meaning:

- <name> - The name of the channel; names are restricted to upper case letters, lower case letters, digits, and the underscore character; no other characters are permitted in a channel name.
- <chid> - The broker assigns a unique, numeric channel id to each channel it maintains; this is formatted as %08lu in requests and responses.
- <message> - The message to post to the channel and to be delivered to all subscribers at the time the message is received; since '|' is used in the queries to separate fields, the message text can consist of all possible characters BUT '|' - i.e., '|' cannot appear in a message.
- <clid> - this is an id that the client application assigns to this subscription; when the server sends a notification to a subscriber, the notification is formatted as <clid>|characters in the message
- <host> - this is a string that has the IP address of the client so it can be called back.
- <service> - this is a string for the service within the client that should be called when the event occurs.
- <port> - this is an unsigned short for the UDP port at which the service in the client is listening.
- <svid> - this is a unique numeric id assigned to the subscriptions by the server, and is returned upon receipt of a successful subscription request.

In this version of PSB, you need to check that the first word in each request is one of the seven legal strings ("CreateChannel", "DestroyChannel", "ListChannels", "ListSubscribers", "Publish", "Subscribe", "Unsubscribe") **and** that the number of arguments is correct. If both are true, `sprintf(response, "1%s", query);` otherwise,
`sprintf(response, "0%s", query);`

3 PSB implements the CreateChannel, ListChannels, ListSubscribers, Subscribe, Publish, and Unsubscribe commands

This version of PSB must correctly create any channels enumerated in the "-f filename" argument at startup and correctly process the CreateChannel, ListChannels, ListSubscribers, Subscribe, Publish, and Unsubscribe commands. You will need to create appropriate data structures (I suggest exploiting one or more of the ADTs from CS 212), store

information in appropriate data structures, and perhaps create additional threads to help you with the functionality.

Building on psbv2, I would implement and test in the following order[3]:

1. implement CreateChannel and modify your code that processes the "-f filename" argument to invoke the CreateChannel code; feel free to log on stderr calls to your code that creates channels
2. implement ListChannels; the Makefile provided in the starting archive builds a test program named listchannels; by invoking "./psbv3 -f filename" in one bash window, you can invoke listchannels in another bash window to test your ListChannels implementation
3. implement Subscribe and Unsubscribe; the Makefile provided in the starting archive builds a test program named subscriber; by invoking "./psbv3 -f filename" in one bash window, you can invoke subscriber in another bash window to test your Subscribe implementation; note that when you send SIGINT to subscriber, the handler routine issues an appropriate Unsubscribe request to your broker.
4. implement ListSubscribers; the Makefile provided in the starting archive builds a test program named listsubscribers; by invoking "./psbv3 -f filename" in one bash window and subscriber one or more times in another bash window, you can invoke listsubscribers in yet another bash window to test your ListSubscribers implementation
5. implement Publish; the Makefile provided in the starting archive builds a test program named publisher; by invoking "./psbv3 -f filename" in one bash window and subscriber one or more times in another bash window, you can invoke publisher in yet another bash window to test your Publish implementation

What are the appropriate data structures to use? Your code needs to be able to associate a channel id with a name; it needs to associate a structure containing information about a channel with a channel id; it needs to associate a structure containing information about a subscriber with a subscriber id. When publishing a message to a channel, your code must determine if there are any subscribers to that channel; if so, your code must send each message to each subscriber.

When you process a subscribe request, you will associate with that subscription a unique subscription id; this is necessary for you to be able to process an unsubscribe request if the

[3] You should feel free to print debug messages on stderr in your implementation. I would appreciate it if each such invocation of fprintf() was sandwiched between #ifdef and #endif lines as in the following:

```
#ifdef DEBUG
fprintf(stderr, …);
#endif /* DEBUG */
```

The CFLAGS macro in the Makefile should have " -DDEBUG" added to it to cause the debug messages to be compiled into your code. Since the debug messages are conditional on DEBUG being defined, it means that I can test and assess your code without seeing all of your debug messages.

client requests that action. Information about the subscriber may have found its way into a data structure that does not permit random access, like a Queue. Since all you can do to a Queue is dequeue() the head element, you will need a member in the subscription structure that is true if the subscription has been cancelled. That way, when the structure is dequeued from the Queue, if it has been cancelled, you will just recycle any heap-allocated storage associated with it and remove the presence of that structure from any other containers.

If you are using separate threads to assist you in the delivery of messages to subscribers, you will likely need to access the shared data structures in [conditional] critical sections.

For all of these commands, the 1st byte of the reply should be '1' if you were able to successfully process the request or '0' if there was an error. The information that follows the success status byte for each of the commands is as follows:

- CreateChannel - sprintf(reply, "1%lu", channelId);
- ListChannels - reply = "1name[,name]*"; /* name is a channel name */
- Subscribe - sprintf(reply, "1%lu", subscriptionId);
- Unsubscribe - sprintf(reply, "1%lu", subscriptionId);
- ListSubscribers - reply = "1subInfo[\nsubInfo]*"
  /* subInfo is of the form "host|service|port" */
- Publish - sprintf(reply, "1%lu", channelId);

In all cases, if there is an error, the reply for the query contained in query is generated using sprintf(reply, "0%s", query);

For example, if the broker receives a query of the form
"Publish|Channel1|hi there"
and Channel1 has a single subscription with host, service, and port, the following pseudocode shows how your code will send the message to the subscriber (obviously the variables used must be appropriately declared and req has the message to send):

```
bxpc = bxp_connect(host, port, service, 1, 1);
reqlen = (unsigned) (strlen(req) + 1);
bxp_call(bxpc, req, reqlen, resp, sizeof resp, &resplen);
bxp_disconnect(bxpc);
```

4 PBS implements the remaining command and focuses on performance

This version must implement the DestroyChannel command. If successful, the reply to the request is generated using sprintf(reply, "1%lu", channelId);

Connecting to and disconnecting from the subscriber is a fairly high overhead way to publish each message to a subscriber, and a solution that caches BXPConnections in some way would be much more efficient.

Subscribers can die without unsubscribing from subscribed channels. This can affect two aspects of message delivery:

- an rpc_connect() call can fail because the subscriber process no longer exists
- an rpc_call() to send a message can fail because the subscriber process no longer exists

It is recommended that when such a situation arises, you flag the subscription as cancelled, just as you do when processing an Unsubscribe request, such that you can clean up these cancelled subscriptions in some other part of your code.

5 The Buffer Exchange Protocol (BXP)

One often encounters the requirement for a simple request/response networking capability. One can craft such a system using TCP, but reliable stream protocols were designed to maximize bulk data throughput. Some RPC systems exist, but require that one define one's interface in an interface definition language, execute a stub compiler to generate client stubs and server skeletons, and run a number of auxiliary services in order to use the system.

This section describes a simple buffer-exchange (BXP) system over UDP; it is derived from a Simple RPC system documented at https://github.com/jsventek/SRPC. Each request/response is in terms of exchanging an outgoing char buffer, and an incoming char buffer, with the server. This document describes the C interface and implementation of this BXP system.

5.1 The C API

The API provided to C programmers is shown below. Both clients and servers invoke the bxp_init() method to initialize the PThread implementation of the architecture shown in Figures 1 and 2 below. The remaining methods are divided into two groups: those used by clients and those used by servers. The following listing shows the API; subsequent subsections show stylized code for a client, a server, and a client supporting a callback service using the API.

5.1.1 The BXP header file

```
/* BSD Header removed to conserve space */

/*
 * bxp - a simple UDP-based Buffer eXchange Protocol system
 */

#ifndef _BXP_H_
#define _BXP_H_

#include "BXP/endpoint.h"

typedef void *BXPConnection;
typedef void *BXPService;

/*
 * initialize BXP system - bind to `port' if non-zero
 * otherwise port number assigned dynamically
 * if `ifEncrypted' is true, prepares the runtime to connect and offer encrypted
```

```
 * connections; if false, can neither create nor offer encrypted connections
 * returns 1 if successful, 0 if failure
 */
int bxp_init(unsigned short port, int ifEncrypted);

/*
 * the following methods are used by BXP clients
 */

/*
 * obtain our ip address (as a string) and port number
 */
void bxp_details(char *ipaddr, unsigned short *port);

/*
 * reverse lookup of ip address (as a string) to fully-qualified hostname
 */
void bxp_reverselu(char *ipaddr, char *hostname);

/*
 * send connect message to host:port with initial sequence number
 * svcName indicates the offered service of interest
 * if ifEncrypted is true, creates an encrypted connection with the server
 * returns 1 after target accepts connect request
 * else returns 0 (failure)
 */
BXPConnection bxp_connect(char *host, unsigned short port, char *svcName,
                          unsigned long seqno, int ifEncrypted);

/*
 * make the next BXP call, waiting until response received
 * upon successful return, `resp' contains `rlen' bytes of data
 * returns 1 if successful, 0 otherwise
 */
int bxp_call(BXPConnection bxpc, void *query, unsigned qlen,
             void *resp, unsigned rsize, unsigned *rlen);

/*
 * disconnect from target
 * no return
 */
void bxp_disconnect(BXPConnection bxpc);

/*
 * the following methods are used to offer and withdraw a named service
 */

/*
 * offer service named `svcName' in this process
 * returns NULL if error
 */
BXPService bxp_offer(char *svcName);

/*
 * withdraw service
 */
void bxp_withdraw(BXPService bxps);

/*
 * the following methods are used by a worker thread in an BXP server
 */

/*
 * obtain the next query message from `bxps' - blocks until message available
 * `len' is the size of `qb' to receive the data
 * upon return, ep has opaque sender information
 *              qb has query data
 *
 * returns actual length as function value
 * returns 0 if there is some massive failure in the system
 */
```

```
unsigned bxp_query(BXPService bxps, BXPEndpoint *ep, void *qb, unsigned len);

/*
 * send the next response message to the `ep'
 * `rb' contains the response to return to the caller
 * returns 1 if successful
 * returns 0 if there is a massive failure in the system
 */
int bxp_response(BXPService bxps, BXPEndpoint *ep, void *rb, unsigned len);

/*
 * the following methods are used to prevent parent and child processes from
 * colliding over the same port numbers
 */

/*
 * suspends activities of the BXP state machine by locking the connection
 * table
 */
void bxp_suspend();

/*
 * resumes activities of the BXP state machine by unlocking the connection
 * table
 */
void bxp_resume();

/*
 * reinitializes the BXP state machine: purges the connection table, closes
 * the original socket on the original UDP port, creates a new socket and
 * binds it to the new port, finally resumes the BXP state machine
 */
int bxp_reinit(unsigned short port);

/*
 * shutdown the BXP system
 */
void bxp_shutdown(void);

#endif /* _BXP_H_ */
```

5.1.2 Client example

```
#include "BXP/bxp.h"
#include <assert.h>
#include <stdio.h>
#include <string.h>
#define UNUSED __attribute__((unused))

int main(UNUSED int argc, UNUSED char *argv[]) {
    BXPConnection bxp;
    char response[10001];
    char request[10000];
    unsigned reqlen, rsplen;
    char buf[BUFSIZ];

    assert(bxp_init(0, 1));
    assert((bxp = bxp_connect("localhost", 5000, "Echo", 1, 1)));
    while (fgets(buf, BUFSIZ, stdin) != NULL) {
        strcpy(request, buf);
        reqlen = strlen(request) + 1;
        assert(bxp_call(bxp, request, reqlen, response, 10000, &rsplen));
        if (response[0] == '0')
            fprintf(stderr, "Error response from server\n");
        else
            fputs(response+1, stdout);
    }
    bxp_disconnect(bxp);
}
```

5.1.3 Server example

```
#include "BXP/bxp.h"
#include <assert.h>
#include <stdio.h>
#include <string.h>
#define UNUSED __attribute__((unused))

int main(UNUSED int argc, UNUSED char *argv[]) {
    BXPEndpoint ep;
    BXPService svc;
    char query[10000], response[10001];
    unsigned qlen, rlen;

    assert(bxp_init(5000, 1));
    assert((svc = bxp_offer("Echo")));
    while ((qlen = bxp_query(svc, &ep, query, 10000)) > 0) {
        sprintf(response, "1%s", query);
        rlen = strlen(response) + 1;
       assert(bxp_response(svc, &ep, response, rlen));
    }
    return 0;
}
```

5.1.4 Client that establishes a callback service

```
#include "BXP/bxp.h"
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#define UNUSED __attribute__((unused))
#define SERVICE "Callback"

void *svcFxn(void *args) {
    BXPService bxps = (BXPService)args;
    BXPEndpoint ep;
    char query[1024], *resp = "1";
    unsigned qlen;

    while ((qlen = bxp_query(bxps, &ep, query, 1024)) > 0) {
        bxp_response(bxps, &ep, resp, 2);
        fputs(query, stdout);
    }
    return NULL;
}

int main(UNUSED int argc, UNUSED char *argv[]) {
    BXPConnection bxp;
    BXPService bxps;
    char response[10001];
    char request[10000];
    unsigned reqlen, rsplen;
    char ipaddr[16];
    unsigned short svcPort;
    pthread_t svcThread;
    char buf[BUFSIZ];

    assert(bxp_init(0, 1));
    bxp_details(ipaddr, &svcPort);
    assert((bxps = bxp_offer(SERVICE)));
    assert((bxp = bxp_connect("localhost", 5000, "Echo-Callback", 1, 1)));
    assert(! pthread_create(&svcThread, NULL, svcFxn, (void *)bxps));
    while (fgets(buf, BUFSIZ, stdin) != NULL) {
        sprintf(request, "%s|%u|%s|%s", ipaddr, svcPort, SERVICE, buf);
        reqlen = strlen(request) + 1;
        assert(bxp_call(bxp, request, reqlen, response, 10000, &rsplen));
        if (response[0] == '0')
            fprintf(stderr, "Error response from server\n");
    }
    bxp_disconnect(bxp);
```

```
}
```

5.1.5 Server that uses call back service of client

```
#include "BXP/bxp.h"
#include <assert.h>
#include <stdio.h>
#include <string.h>
#define UNUSED __attribute__((unused))

int extractWords(char *buf, char *sep, char *words[]) {
    int i;
    char *p;

    for (p = strtok(buf, sep), i = 0; p != NULL; p = strtok(NULL, sep), i++)
        words[i] = p;
    words[i] = NULL;
    return i;
}

int main(UNUSED int argc, UNUSED char *argv[]) {
    BXPEndpoint ep;
    BXPService svc;
    char query[10000], response[10001];
    unsigned qlen, rlen;

    assert(bxp_init(5000, 1));
    assert((svc = bxp_offer("Echo-Callback")));
    while ((qlen = bxp_query(svc, &ep, query, 10000)) > 0) {
        char *words[25];
        char out[BUFSIZ];
        char back[128];
        BXPConnection bxpc;
        unsigned port, backlen;
        (void) extractWords(query, "|", words);
        sscanf(words[1], "%u", &port);
        bxpc = bxp_connect(words[0], (unsigned short)port, words[2], 1, 1);
        strcpy(out, words[3]);
        bxp_call(bxpc, out, strlen(out)+1, back, sizeof back, &backlen);
        bxp_disconnect(bxpc);
        sprintf(response, "1%s", query);
        rlen = strlen(response) + 1;
        assert(bxp_response(svc, &ep, response, rlen));
    }
    return 0;
}
```

5.1.6 Makefile for the previous four programs

```
CFLAGS=-W -Wall -pedantic -I/usr/local/include
LDFLAGS=-L/usr/local/lib
PROGRAMS=client server cbclient cbserver
LIBRARIES=-lBXP -lpthread

all: $(PROGRAMS)

client: client.o
	gcc $(LDFLAGS) -o $@ $^ $(LIBRARIES)

cbclient: cbclient.o
	gcc $(LDFLAGS) -o $@ $^ $(LIBRARIES)

server: server.o
	gcc $(LDFLAGS) -o $@ $^ $(LIBRARIES)

cbserver: cbserver.o
	gcc $(LDFLAGS) -o $@ $^ $(LIBRARIES)

client.o: client.c
cbclient.o: cbclient.c
```

```
server.o: server.c
cbserver.o: cbserver.c

clean:
	rm -f $(PROGRAMS) *.o
```

5.2 General Architecture

Each process wishing to use the BXP system consists of at least three threads:

- a receiving thread that retrieves UDP packets from the network, and places each one in an incoming queue
- a timer thread that processes retries of certain messages as well as being responsible for purging stale connections
- to provide a service, one or more worker threads; each such thread obtains an element from an incoming queue, performs some actions, and sends the response back to the requestor
- to use a service, one or more client threads; each such thread issues blocking requests to the selected service

The sending thread, timer thread, and worker threads sending a response also share connection state for each active connection.

The architecture in a server is displayed in Figure 1 below. The architecture in a client is shown in Figure 2.

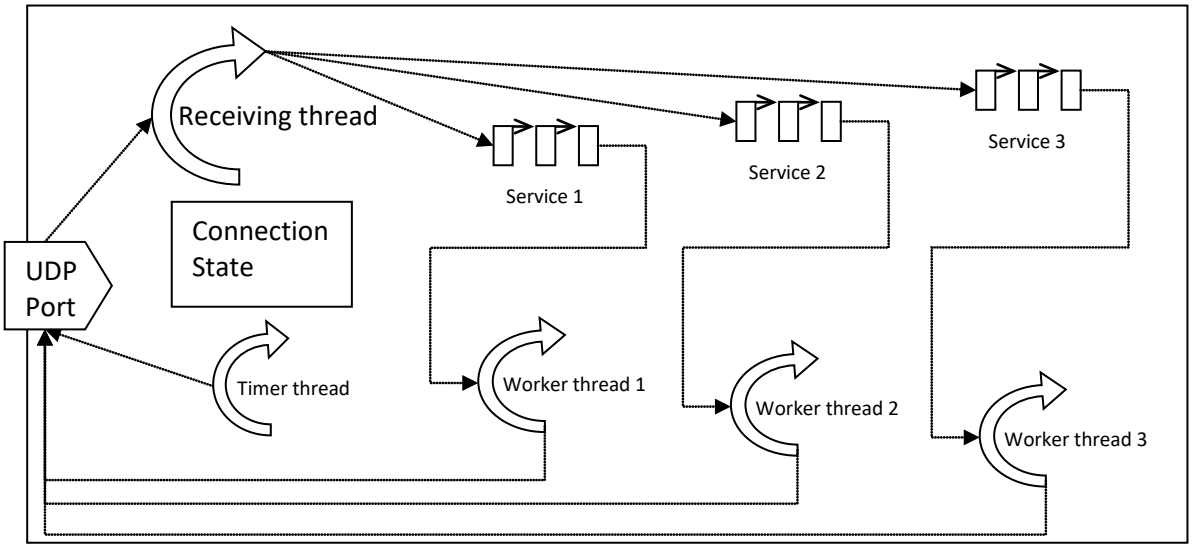


Figure 1: Server architecture of the Buffer eXchange Protocol system

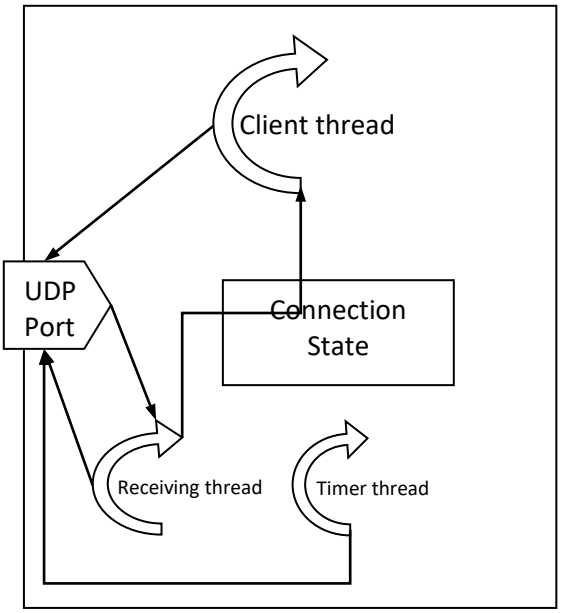


Figure 2: Client architecture of the Buffer eXchange Protocol system

5.3 Connection States

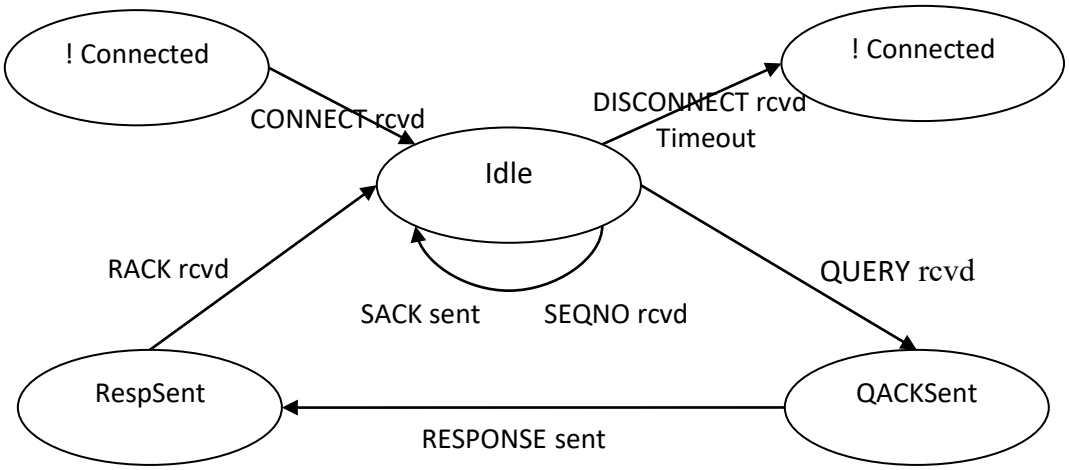
BXP communication is basically request/response. There is no absolute requirement to maintain connection state, but it does make it much easier to provide the best-effort semantics embodied in BXP interactions; additionally, it tends to make the system easier to build and maintain. And, finally, it provides the opportunity to successfully secure the connection using a combination of asymmetric and symmetric encryption.

Each process in an internet host is associated with one or more UDP ports. There is no technical reason to have more than one port, so the remainder of the discussion, as well as Figures 1 and 2 above, is in terms of a single port for a process.

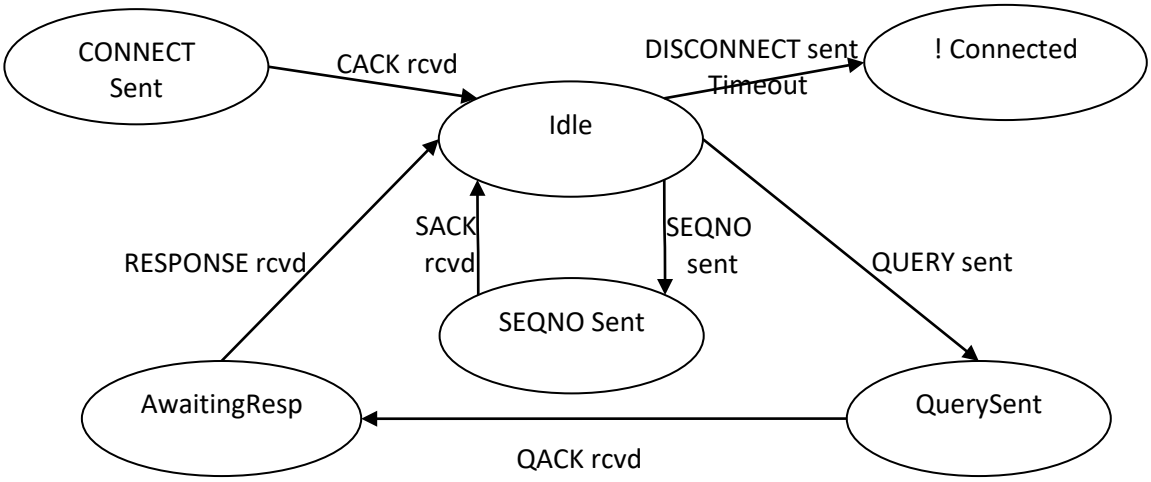
In general, a triple of identifiers is required to uniquely identify each connection:

- the IP address of the host upon which the process is running
- the UDP port number associated with the running process
- a multiplexing identifier within the process, usually one per active thread

Each connection goes through the following life cycle on the server side of a connection:



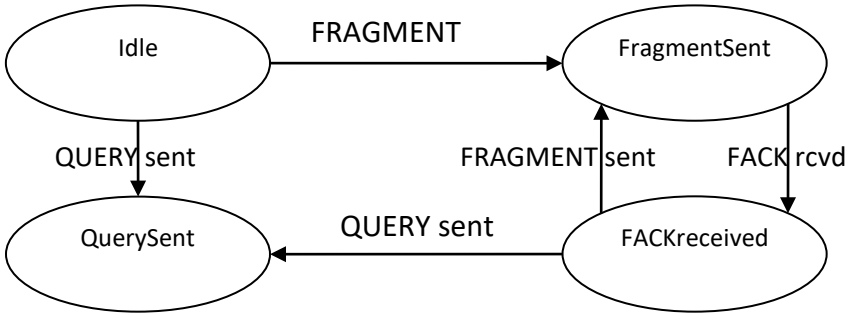
The equivalent life cycle diagram on the client side of a connection is as follows:



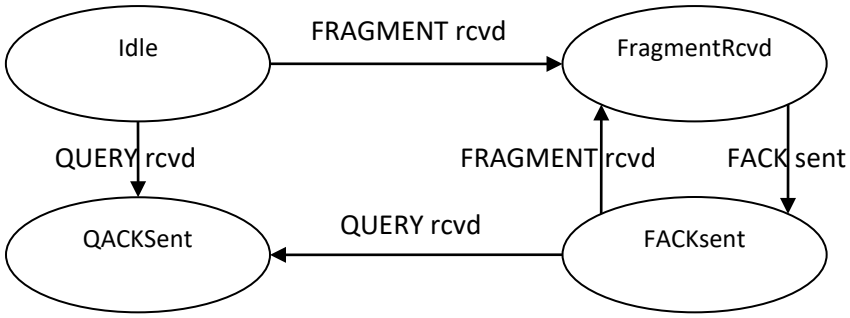
5.4 Fragmentation and Reassembly states

If the query or response buffer exceeds the size of a single UDP datagram, the buffer must be broken up into fragments and reassembled at the receiving end. Therefore, the actions labelled “QUERY sent”, “QUERY rcvd”, “RESPONSE sent”, and “RESPONSE rcvd” in the state diagrams above are actually nested state machines.

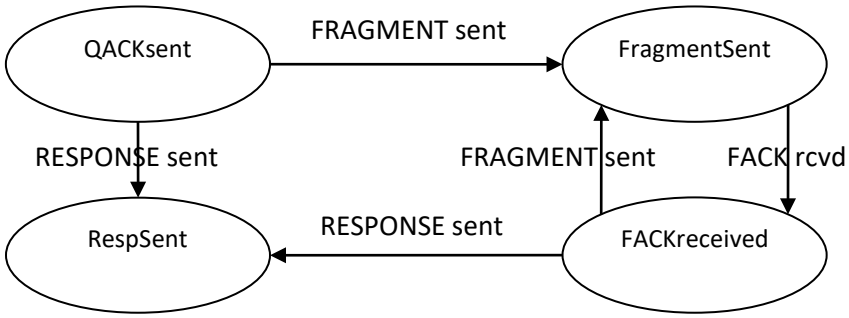
5.4.1 QUERY sent



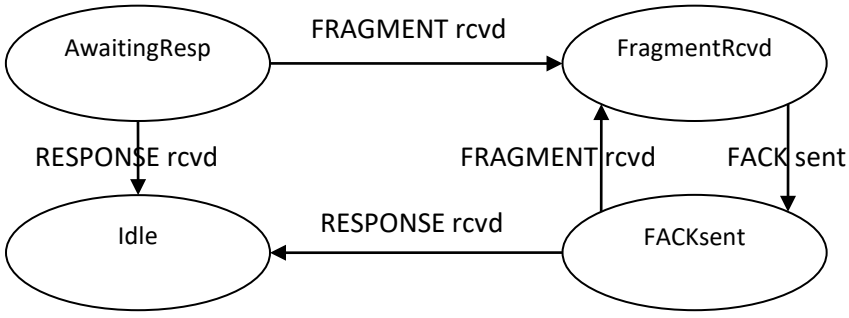
5.4.2 QUERY rcvd



5.4.3 RESPONSE sent



5.4.4 RESPONSE rcvd



5.5 Protocol Message Types

A number of protocol message types are shown as labels on the transitions in the life cycle diagrams in the previous section. Each of these is enumerated here, with pseudocode for the action taken by the state machine receiving such a message. Initially, we focus on messages sent by both requestors and responders. Then we focus on the messages sent by a requestor and received by a responder. Finally, we will look at the opposite direction.

N.B. You may ignore sections 5.5.1-5.5.3; they are provided for those students who may have an interest in what goes into a transport level protocol.

5.5.1 Both directions

PING — Once a connection is established, connection state is kept by each participant. Participants have a habit of occasionally disappearing without gracefully terminating the connection. It is important that each participant is able to determine if its correspondent is still alive, terminating the connection if the answer to this question is in the negative. This is done using pings and ping acknowledgments to construct a failure detector.

The receiving state machine does the following upon receipt of a PING message:

- If it has a connection record corresponding to that client identifier, it sends a PACK (Ping ACK) response to the sender.
- If not, it drops the message

PACK — The receiving state machine does the following upon receipt of a PACK message:

- If it has a connection record corresponding to that client identifier, it resets the `pingsTilPurge` and `ticksTilPing` fields of the connection record
- If not, it drops the message.

5.5.2 Requestor → Responder

CONNECT — Before a client makes a request of a server, it must first explicitly connect with the server. The client sends a CONNECT protocol message, which provides the client identifier [triple] to the server for naming the connection and an initial value for the sequence number that will be used for this connection.

The receiving state machine does the following upon receipt of a CONNECT message:

- if it does NOT have any state associated with that client identifier, it creates a connection record for that client, marks it as IDLE, and sends a CACK (Connection ACK) response to the client
- if it already has a connection record for that client, and it is in the IDLE state, it assumes that the CACK was lost, and resends it
- if it already has a connection record for that client, and it is NOT in the IDLE state, it simply ignores the message, as the client state machine is obviously confused

SEQNO The requestor can reset the sequence number at any time when the connection is in the IDLE state.
The receiving state machine does the following upon receipt of a SEQNO message:

- if it does NOT have any state associated with that client identifier, it drops the message, as the client state machine is confused
- if it already has a connection record for that client, and it is in the IDLE state, it resets the sequence number to the value supplied in the SEQNO protocol message, and replies with a SACK message
- if it is in the RESPONSE_SENT state, it assumes that the RACK message was not delivered, it sets the connection state to IDLE, and then acts according to the previous bullet
- if it already has a connection record for that client, and it is NOT in the IDLE or RESPONSE_SENT state, it simply ignores the message, as the client state machine is obviously confused

QUERY Clients issue queries over connections, expecting responses. The QUERY protocol message conveys a request for which the server is expected to provide a response.
The receiving state machine does the following upon receipt of a QUERY message:

- if it does NOT have a connection record for that client identifier, it drops the message, as the client state machine is obviously confused
- if the connection record is in the IDLE state, the next actions depend upon the relationship between the sequence number in the received query and the sequence number stored in the connection record:
    - query number <= record number
      this is a replay of previously answered query; client state machine is confused, message is dropped
    - query number > record number + 1
      gaps are not allowed in sequence numbers, so this is illegal; again, client state machine is confused, message is dropped
    - have next legal query on this connection; takes the following steps
        - increment connection record sequence number
        - set state to QACKSent
        - send the QACK
        - place query in appropriate service queue
- record is in the FACKSent state; action depends upon relationship between query sequence number and record sequence number and the fragment number
    - if query sequence number != record sequence number, client state machine is confused, drop message
    - if the fragment number is one more than the last received fragment, this QUERY message completes the buffered query; take the following actions:
        - append data to the previously allocated buffer
        - set state to QACKSent

        - send the QACK
        - place the completed buffer in the appropriate service queue
    - if the fragment numbers are equal, FACK was lost, resend FACK
- record is in the QACKSent state; again, action depends upon relationship between query sequence number and record sequence number
    - if query sequence number != record sequence number, client state machine is confused, drop message
    - QACK was lost, resend QACK
- record is in the RespSent state; as before, sequence number relationships are important
    - query sequence number == record sequence number – apparently, client NEVER received QACK, nor has it received the Response that we sent causing us to be in the RespSent state; resend the response, remaining in this state
    - query sequence number == record sequence number + 1 – client has received the response, but we have not received the RACK acknowledging receipt of the previous response; change the state of the connection record to IDLE, then process this QUERY as described for the IDLE state above
    - any other query sequence number - client state machine is confused, so drop the message

FRAGMENT If the query is too large to fit into a single UDP packet, and is broken up into n-1 fragments, with the last fragment sent as a QUERY message. The FRAGMENT protocol message conveys a fragment that must be reassembled by the responder.

The receiving state machine does the following upon receipt of a FRAGMENT message:

- if it does NOT have a connection record for that client identifier, it drops the message, as the client state machine is obviously confused
- if the connection record is in the IDLE state, the next actions depend upon the relationship between the sequence number in the received fragment and the sequence number stored in the connection record, as well as the fragment number:
    - query number <= record number
      this is a replay of previously answered query; client state machine is confused, message is dropped
    - query number > record number + 1
      gaps are not allowed in sequence numbers, so this is illegal; again, client state machine is confused, message is dropped
    - if the fragment number is 1, we have the first fragment of the next legal query on this connection; take the following steps
        - increment connection record sequence number
        - allocate buffer to hold ENTIRE query
        - copy packet data into beginning of that buffer
        - set state to FACKSent
        - send the FACK

- record is in the FACKSent state; again, action depends upon relationship between fragment sequence number and record sequence number
    - if query sequence number == record sequence number and the fragment numbers in the packet and the record are equal, the FACK was lost, resend the FACK
    - if the sequence numbers are equal, and the packet fragment number is one larger than the last sent, then we have received the next fragment; take the following steps:
        - copy fragment data into query buffer
        - set state to FACKSent
        - send the FACK
    - client state machine is confused, drop the message
- record is in the RespSent state; as before, sequence number relationships are important
    - query sequence number == record sequence number + 1 and fragment number is 1 – client has received the response, but we have not received the RACK acknowledging receipt of the previous response; change the state of the connection record to IDLE, then process this FRAGMENT as described for the IDLE state above
    - any other query sequence number or fragment number - client state machine is confused, so drop the message

RACK The RACK protocol message acknowledges receipt of the response previously sent.

The receiving state machine does the following upon receipt of a RACK message:

- if it does NOT have a connection record for that client identifier, it drops the message, as the client state machine is obviously confused
- if the record is in the IDLE state, the client state machine is confused, and the message is dropped
- if record is in the QACKSent state, the client state machine is confused, and the message is dropped
- record is in the RespSent state; actions depend upon the sequence number relationship
    - query sequence number == record sequence number – normal behaviour, the client is acknowledging receipt of the response; change the state of the connection record to IDLE
    - any other query sequence number - client state machine is confused, so drop the message

DISCONNECT The DISCONNECT protocol message declares the client's desire to terminate the connection.

The receiving state machine does the following upon receipt of a DISCONNECT message:

- it always generates a corresponding DACK and sends it back
- if it has a connection record, it sets the connection record state to TIMEDOUT

5.5.3 Responder → Requestor

CACK When the user invokes bxp_connect(), a connection record is created, recording the specified starting sequence number and setting the state to CONNECTSent.

The receiving state machine does the following upon receipt of a CACK message:

- if it does NOT have any state associated with that server identifier, it drops the message, as the server state machine is confused
- if it already has a connection record for that client, and it is in the CONNECTSent state, it changes the state to IDLE and returns to the calling client
- if it already has a connection record for that client, and it is in some other state, it simply ignores the message, as the server state machine is confused

SACK After sending a SEQNO, the client state machine expects to receive a SACK message.

The receiving state machine does the following upon receipt of a SACK message:

- if it does NOT have any state associated with that server identifier, it drops the message, as the server state machine is confused
- if it is in the IDLE state, it drops the SACK message, as the server state machine is confused
- If it is the SeqnoSent state, it sets the connection state to IDLE

QACK After sending a QUERY, the client state machine expects to receive a QACK message.

The receiving state machine does the following upon receipt of a QACK message:

- if it does NOT have any state associated with that server identifier, it drops the message, as the server state machine is confused
- if it is in the IDLE state, it drops the QACK message, as the server state machine is confused
- If it is the QuerySent state, action depends upon the relationships between the sequence numbers
    - if the sequence numbers are identical, then change the record state to AwaitingResponse
    - for any other sequence numbers, drop the message as the server state machine is confused
- if it is in the AwaitingResponse state, drop the message

RESPONSE After processing the QUERY, the server sends a RESPONSE message.

The receiving state machine does the following upon receipt of a RESPONSE message:

- if it does NOT have any state associated with that server identifier, it drops the message, as the server state machine is confused

- if it is in the IDLE state, it drops the message, as the server state machine is confused
- If it is the QuerySent state, action depends upon the relationships between the sequence numbers
    - if the sequence numbers are identical, then it appears that we never received a QACK message, but have received the RESPONSE corresponding to the outstanding QUERY; change the record state to IDLE, send the RACK, and enable delivery of the response to the caller
    - for any other sequence numbers, drop the message as the server state machine is confused
- if it is in the AwaitingResponse state, and the sequence numbers are identical, change the record state to IDLE and enable delivery of the response to the caller; otherwise, drop the message
- if it is in the FACKSent state and this is the last fragment of the response and the sequence numbers are identical, append the message data to the previously allocated buffer, change record state to IDLE, send the RACK, and enable delivery of the response to the caller

FRAGMENT If the response is too large to fit into a single UDP packet, and is broken up into n-1 fragments, with the last fragment sent as a RESPONSE message. The FRAGMENT protocol message conveys a fragment that must be reassembled by the requestor.

The receiving state machine does the following upon receipt of a FRAGMENT message:

- if it does NOT have a connection record for that client identifier, it drops the message, as the client state machine is obviously confused
- if the connection record is in the AwaitingResponse state, the next actions depend upon the relationship between the sequence number in the received fragment and the sequence number stored in the connection record, as well as the fragment number:
    - if the sequence numbers are equal, and the fragment number is 1, we have the first fragment of the next legal response on this connection; take the following steps
        - allocate buffer to hold ENTIRE response
        - copy packet data into beginning of that buffer
        - set state to FACKSent
        - send the FACK
- record is in the FACKSent state; again, action depends upon relationship between fragment sequence number and record sequence number
    - if query sequence number == record sequence number and the fragment numbers in the packet and the record are equal, the FACK was lost, resend the FACK
    - if the sequence numbers are equal, and the packet fragment number is one larger than the last sent, then we have received the next fragment; take the following steps:
        - copy fragment data into response buffer

        - set state to FACKSent
        - send the FACK
    - otherwise, client state machine is confused, drop the message
- record is in the QuerySent state; as before, sequence number relationships are important
    - query sequence number == record sequence number and fragment number is 1 – we somehow missed the QACK for the current QUERY, and this is the first fragment of the response; change the state of the connection record to AwaitingResponse, then process this FRAGMENT as described for the AwaitingResponse state above
    - any other query sequence number or fragment number - client state machine is confused, so drop the message

DACK After sending a DISCONNECT, the client state machine expects to receive a DACK message.

The receiving state machine does the following upon receipt of a DACK message:

- if it does NOT have any state associated with that server identifier, it drops the message, as the server state machine is confused
- Otherwise, it sets the state of the connection to TIMEDOUT

5.6 Persistence, timeouts, and other miscellany

UDP does not provide guaranteed delivery of packets, so any reliable protocol constructed over UDP must implement mechanisms to mask transient delivery failures of individual packets. This is accomplished using two devices in this protocol:

- As may be evident from the discussion in the previous section, the receiving state machines attempt to be very flexible in what they accept; in particular, there are a couple of situations where even though the explicit acknowledgement scheme is violated, the protocol is able to function correctly in the absence of these particular ACK messages.
- Successful message delivery (as determined by explicit ACKs or the short circuits discussed in the previous bullet) is facilitated by attempting retransmission of each outbound packet according to an exponentially increasing time schedule. After an implementation-defined maximum number of retransmission attempts, the implementation will assume that the failure is not transient, and will set the state of the connection to TIMEDOUT.
- If a PACK is not received after an implementation-defined number of PINGs, the connection state is set to TIMEDOUT. Note that regular messaging activity on the connection always resets the failure detector fields in the connection record, so PINGs are only sent if the connection has been idle (not necessarily in the ST_IDLE state ☺).
- The sequence number sequence will need to wrap; the responsibility for resetting the sequence number is with the client/requestor; when it determines that it must reset the sequence number, it sends a SEQNO protocol message, expecting a SACK response.
- All harvesting of stale connections is done by the timer thread; the next timer pass after a connection has been set to TIMEDOUT will remove such connection records from the table and return any allocated resources.

6 Starting archives

6.1 P3start.tgz

This archive contains the following files:

.psbrc	A data file that you can use to test out the -f flag for psbv[1-4].
echoclient.c	A client that you can use to test out psbv1.c and psbv2.c.
v2test.dat	A data file that you can use with echoclient to test out psbv2.c.
listchannels.c	A client that you can use to test out psbv3.c and psbv4.c.
listsubscriptions.c	A client that you can use to test out psbv3.c and psbv4.c.
subscriber.c	A client that you can use to test out psbv3.c and psbv4.c.
publisher.c	A client that you can use to test out psbv3.c and psbv4.c.
Makefile	Has targets to compile and link echoclient, listchannels, listsubscribers, subscriber, publisher, psb[1-4].
sort.h	Header file for a sort implementation
sort.c	Implementation of the function defined in the header file.

6.2 Installing BXP

P3BXP-INTEL.tgz and P3BXP-ARM.tgz are available on Canvas. You should download the archive appropriate to your architecture; extract the contents of the archive - it will create a bxp directory in your current directory when you invoke tar. cd into bxp, then issue the following three commands to bash:

```
% sudo make installheaders
% sudo make installlibrary
% sudo make installmanpages
```

YOU MUST INSTALL BXP AS DESCRIBED ABOVE.

6.3 Installing the ADT headers, library, and man pages

P3ADTS-INTEL.tgz and P3ADTS-ARM.tgz are also available on Canvas. You should download the archive appropriate to your architecture; extract the contents of the archive - it will create an adts directory in your current directory when you invoke tar. cd into adts, then issue the following three commands to bash:

```
% sudo make installheaders
% sudo make installlibrary
% sudo make installmanpages
```

YOU MUST INSTALL ADTS AS DESCRIBED ABOVE IF YOU HAVE NOT ALREADY DONE SO.

7 Developing Your Code

The best way to develop your code is in Linux running inside the virtual machine image provided to you. This way, if you crash the system, it is straightforward to restart. This also gives you the benefit of taking snapshots of system state right before you do something potentially risky or hazardous, so that if something goes horribly awry you can easily roll back to a safe state.

You should use your Bitbucket GIT repositories for keeping track of your programming work. As a reference, you can perform the command line steps below to create a new project directory and upload it to your uoregon-cis415 repository.

```
% cd /path/to/your/uoregon-cis415
% mkdir project3
% echo "This is a test file." >project1/testFile.txt
% git add project3
% git commit –m "Initial commit of project3"
% git push –u origin master
```

Any subsequent changes or additions can be saved via the add, commit, and push commands.

7.1 Testing Your Code

In P3start.tgz, I have provided the source for several programs that you can use to test your server: echoclient.c, listchannels.c, listsubscribers.c, subscriber.c and publisher.c; echoclient can be used to test dtsv1 and dtsv2, while the others can be used to test dtsv3 and dtsv4. You will have to read over the source code for the provided test programs to see how to invoke them. I have provided a Makefile in the archive that will build these two programs; it also has targets to build psbv1, psbv2, psbv3, and psbv4. I have also provided sort.h and sort.c since the response to the ListChannels command is supposed to have the Channel names in alphabetical order.

7.2 Checking for memory leaks in your code

The buffer exchange protocol uses malloc()/free() quite a bit while implementing the state machines shown in sections 5.3 and 5.4 above. In addition, your server should run forever, so you have to terminate it when running it under valgrind, thus making the summary message you receive when you ^C out of the server less than useful.

valgrind enables you to display a leak check summary within your program by doing the following:

1. At the top of your source file, add "#include <valgrind/valgrind.h>" to the list of included files.
2. Wherever you wish to generate a leak check summary, include the following statement
   VALGRIND_MONITOR_COMMAND("leak_check summary");

If you are not running your program under valgrind, this statement will not generate any output. If you are running your program under valgrind, a leak summary will be generated on standard error each time you execute this statement. The leak summary will look like the following:

```
==pid#== LEAK SUMMARY:
==pid#==    definitely lost: 0 (+0) bytes in 0 (+0) blocks
==pid#==    indirectly lost: 0 (+0) bytes in 0 (+0) blocks
```

```
==pid#==      possibly lost: 1,088 (+0) bytes in 4 (+0) blocks
==pid#==    still reachable: 5,952 (+124) bytes in 35 (+5) blocks
==pid#==         suppressed: 0 (+0) bytes in 0 (+0) blocks
==pid#== To see details of leaked memory, give 'full' arg to leak_check
==pid#==
```

How can you use this feature in psbv3 to check for your memory leaks? I suggest the following:

- After you have created all of the ADT instances you need, have initialized BXP, created the thread to process the requests, and that thread has offered the "DSP" service - this establishes the heap memory baseline.
- After you have processed each client request received (either determined there is an error, or processed the request).
- After you have completed each Publish request. (this assumes that you are delegating the process of delivering each message to a channel's subscribers to a separate thread working from a Queue

If the "definitely lost" line of each summary continues to read 0 for each of these summaries, your processing is not leaking any memory. Since this server is supposed to run forever, it is especially important that your code leak no memory. This is how I will be assessing the "No memory leaks or error messages" in the grading rubric.

7.3 Helping your Classmate

This is an individual assignment. You should be reading the manuals and man pages, hunting for information, and learning those things that enable you to do the project. However, it is important for everyone to make progress and hopefully obtain the same level of knowledge by the project's end. If you get stuck, seek out help to get unstuck. Sometimes just having another pair of eyes looking at your code is all you need. If you cannot obtain help from the TA, the LAs, or the instructor, it is possible that a classmate can be of assistance.

In your status report on the project, you should provide the names of classmates that you have assisted, with an indication of the type of help you provided. You should also indicate the names of classmates from whom you have received help, and the nature of that assistance.

Each of your source files must start with an "authorship statement", contained in C comments, as follows:

state your name, your login, and the title of the assignment (CIS 415 Project 3)

state either "This is my own work." or "This is my own work except that …", as appropriate.

Note that this is not a license to collude. We will be checking for collusion; better to turn in an incomplete solution that is your own than a copy of someone else's work. We have very good tools for detecting collusion.

8 Submission[4]

You will submit your solutions electronically by uploading a gzipped tar archive via Canvas.

Your TGZ archive should be named <duckid>-project3.tgz, where <duckid> is your "duckid". It should contain the C source files you are submitting for assessment - do NOT submit dummy versions of the BSP server, just those you have worked on. Besides psbv?.c, you must provide a Makefile for creating the associated executables named psbv?, and a document named report.txt, describing the state of your solution, and documenting anything of which we should be aware when marking your submission. If you have created any additional .c or .h files upon which psbv[1-4] depend, be sure to include them in the archive, as well. Note that the Makefile **SHOULD NOT** have targets for echoclient, listchannels, listsubscriptions, publisher, or subscriber.

Within the archive, these files **should not be contained** in a folder. Thus, if I upload "jsventek-project3.tgz", then I should see something like the following when I execute the following command:

```
$ tar –ztvf jsventek-project1.tgz
-rw-r--r-- jsventek/group     1021 2016-10-30 16:37 Makefile
-rw-r--r-- jsventek/group     3670 2016-10-30 16:30 psbv1.c
-rw-r--r-- jsventek/group     5125 2016-10-30 16:37 psbv2.c
-rw-r--r-- jsventek/group     6531 2016-10-30 16:37 psbv3.c
-rw-r--r-- jsventek/group     8127 2016-10-30 16:37 psbv4.c
-rw-r--r-- jsventek/group     1536 2016-10-30 16:30 report.txt
```

as well as any other files that you have included.

Each of your source files must start with an "authorship statement", contained in C comments, as follows:

- state your name, your duckid, and the title of the assignment (CIS 415 Project 3)
- state either "This is my own work." or "This is my own work except that …", as appropriate.

We will be compiling your code and testing against unseen commands. We will also be checking for collusion; better to turn in an incomplete solution that is your own than a copy of someone else's work. We have very good tools for detecting collusion.

[4] ***A 75% penalty will be assessed if you do not follow these submission instructions. See the handout for Project 0 if you do not remember how to create the gzipped tar archive for submission.***

Rubric for CIS 415, Project 3

Your submission will be marked on a 100 point scale. I place substantial emphasis upon **WORKING** submissions, and you will note that a large fraction of the points is reserved for this aspect. It is to your advantage to ensure that whatever you submit compiles, links, and runs correctly. The information returned to you will indicate the number of points awarded for the submission.

You must be sure that your code works correctly on your virtual machine, regardless of which platform you use for development and testing. Leave enough time in your development to fully test on the virtual machine before submission.

Version	Points	Description
(10)	10	Your report – honestly describes the state of your submission
PSB v1 (20)	8	for workable solution initializes BXP runtime create a thread to receive requests from echoclient • offer PSB service • forever - fetch query from service - format correct response - send response back to client
	1	if it successfully compiles
	1	if it compiles with no warnings
	1	if it successfully links
	1	if it links with no warnings
	6	if it works correctly
	2	if there are no memory leaks
BSP v2 (20)	8	for workable solution Correctly parses query strings echoes back correct query strings with success status byte responds with failure status byte if incorrect command or incorrect number of additional arguments.
	1	if it successfully compiles
	1	if it compiles with no warnings
	1	if it successfully links
	1	if it links with no warnings
	6	if it works correctly
	2	if there are no memory leaks

Version	Points	Description
PSB v3 (50)	20	for workable solution correct data structures correct implementation of CreateChannel correct implementation of ListChannels correct implementation of Subscribe correct implementation of ListSubscribers correct implementation of Unsubscribe correct implementation of Publish
	1	if it successfully compiles
	1	if it compiles with no warnings
	1	if it successfully links
	1	if it links with no warnings
	20	if it works correctly
	6	if there are no memory leaks
PSB v4 (20)	8	for workable solution - working version of pwbv3 PLUS correct implementation of DestroyChannel caches BXPConnections handles subscribers that disappear
	1	if it compiles with no warnings
	1	if it links with no warnings
	8	if it works correctly
	2	if there are no memory leaks