dQUOBEC Event Channel Communication System

Nithya Vijayakumar, Beth Plale
Indiana University
Bloomington, IN
{nvijayak, plale}@cs.indiana.edu

IU-CS TR614

Abstract

We introduce dQUOBEC, the event channel communication system used by dQUOB. It is a publish-subscribe system implemented as event channels and uses Portable Binary Input Output (PBIO) for data transfer. It supports typed channels, in that a channel transfers events having the same type. dQUOBEC has been implemented with C++ and provides C++ and Java API for applications. The main features of dQUOBEC are light weight implementation, easy to use API, decentralized connection handling and efficient memory management. In this paper we discuss our motivation to build dQUOBEC, its design, architecture, scalability and performance.

1. Introduction

A publish-subscribe system provides a loose form of interaction in large-scale settings where publishers provide information in the form of events and subscribers register their interest in a topic or a pattern of events. Registered subscribers asynchronously receive events matching their interest regardless of the events' publisher. The strength of an event-based interaction style is drawn from full decoupling in time, space and flow between publishers and subscribers. In this technical report, we introduce our light weight and efficient event channel communication system, dQUOBEC. dQUOBEC gets its name from dQUOB the primary user application of this event channel system. dQUOB [1] is a middleware for executing continuous queries on data streams. dQUOBEC is designed to be the underlying publish-subscribe system used by dQUOB to transfer binary data as streams.

The rest of this technical report is organized as follows. In section 2, we discuss the event channel requirements of dQUOB. We compare the features of a few existing publish-subscribe systems and discuss our motivation to build dQUOBEC in section 3. Section 4 provides an overview of dQUOBEC. Performance analysis of the system is discussed in section 5. Section 6 describes the future work. A brief overview of the dQUOBEC API and a few examples are provided in the Appendix.

2. dQUOB Requirements

Publish-subscribe systems differ on a number of different characteristics [9]. The most popular decomposition of publish-subscribe systems is into the general categories of *subject based* or *content based* systems. In subject based systems, subscription targets a group or channel and the user receives all events that are associated with that group. Brokering a connection between a publisher and subscriber is the act of connecting a channel supplier with a channel consumer. In content based systems, the decision of whom a message is directed to, is made on a message-by-message basis based on a query or predicate issued by a subscriber. The architecture of publish-subscribe systems can be classified into the general categories of *client server* or *peer to peer* depending on how the components (tasks) in the system are connected. Publish-subscribe systems can be push-based or pull-based or both. In push-based model, messages are automatically broadcast to subscribers. In pull-based model, subscribers request data from publishers as and when needed.

dQUOB [1] is built on the assumption of a push-based streaming model in which events are categorized based on subjects and not contents. dQUOB targets high performance real time decision making and supports a peer to peer architecture. It is implemented in C++ and has been applied to scientific applications which are characterized by large event sizes. It transfers data in binary format. Storing and transmitting data in binary format is often desirable both to conserve I/O bandwidth and to reduce storage and processing requirements.

The event channel communication system needed for dQUOB should satisfy the following requirements: subject based publish-subscribe system capable of transferring binary data with inbuilt encoding and decoding functionality; developed using C/C++ to meet the high performance requirements; follows a push based model; supports a peer to peer architecture; and has a low overhead.

3. Related Work

Our group conducted a survey of the existing publish-subscribe systems in [9]. The systems evaluated were classified based on how the data is categorized, system architecture, push or pull model and the language used for implementation. Some of the systems investigated in our survey are discussed below.

- Scribe [7] is a subject based system with a peer to peer model. It uses lookup tables to store and query information. Though it satisfies some of the main requirements of dQUOB, we could not use this system as it has been implemented in Java. dQUOB has very high real time constraints for which Java is not a viable option. Also dQUOB is implemented in C++ and using a Java based event channel system will impose considerable overheads in byte conversions.

- Narada [3] is a content based publish-subscribe system that uses a client server model. This does not satisfy the needs of dQUOB which is built on the assumption of a subject based system.
- XMessages [8] is a hybrid system that supports both content based and subject based distribution of data. But it follows a client server architecture and has been implemented using Java. For the same reasons as pointed out for Scribe, XMessages also cannot be used in dQUOB.
- Echo [4] is a hybrid system that supports a peer to peer architecture. It provides Java, C and C++ APIs. dQUOB was originally built using Echo as its event channel communication system. Echo has many features not needed for dQUOB. Our need for a light weight and simple solution drove the development of dQUOBEC. dQUOBEC credits Echo for much of its architecture.

dQUOBEC is a subject based publish-subscribe system that follows a push based streaming model and supports a peer-to-peer architecture. It is implemented in C++ and provides C++ and Java API to applications. It uses Portable Binary Input Output (PBIO) [2] for binary data transfer. PBIO is a self describing binary encoding tool used for storing and transmitting binary data. A sender registers the structure of the data that he/she wishes to send with a separate server so that the third parties cannot interpret the result. PBIO then transparently masks the differences in bit ordering, machine word sizes, locations and basic types of fields in the records exchanged.

4. System Overview

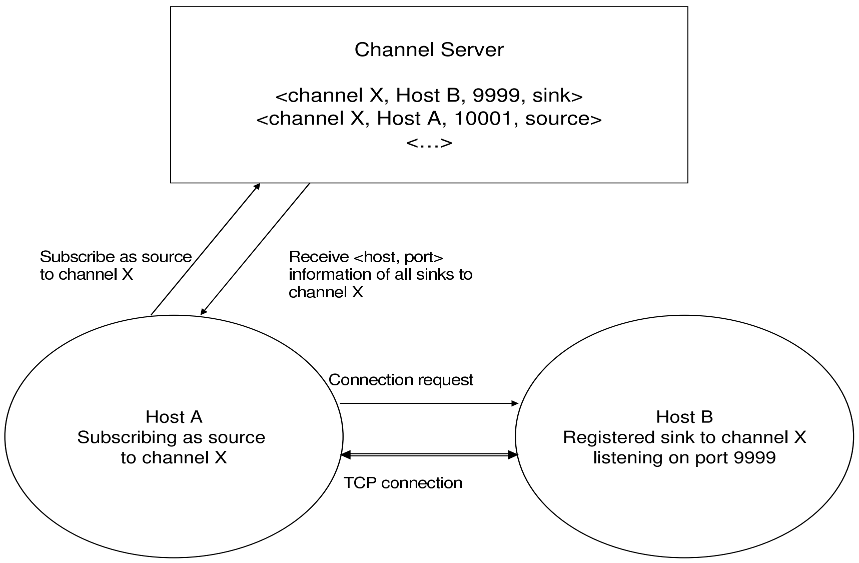
Channel Server
<channel X, Host B, 9999, sink>
<channel X, Host A, 10001, source>
<...>
Subscribe as source to channel X
Receive <host, port> information of all sinks to channel X
Connection request
TCP connection
Host A Subscribing as source to channel X
Host B Registered sink to channel X listening on port 9999

Figure 1. Subscription process in dQUOBEC

dQUOBEC is implemented as a publish-subscribe system. A publisher is often referred to as a “source” and a recipient as a "sink". A channel is a logical entity through which one or more sources send data to all its sinks. dQUOBEC supports typed channels, in that a channel transfers events having the same type. The two main components of dQUOBEC are the channel server and the logical channels. The channel server shown in figure 1, is a repository containing <channel name, host name, port, subscription type> information for all registered subscribers. This information is queried based on channel name. The source and sink are application tasks that use dQUOBEC. A *source* or a *publisher* generates events for one or more channels. A *sink* or a *receiver* receives events from one or more channels. At any time, a task can act as a source and/or sink for multiple channels.

A channel is a logical entity. It is implemented as a point-to-point TCP connection between the publishers (sources) and the subscribers (sinks). Figure 1 shows the procedure adopted by dQUOBEC to establish a connection between a publisher and a subscriber. When a task registers with dQUOBEC as a source/sink for a particular channel, a request message containing the <channelname, hostname, port, subscriptionType> information of this task is sent to the channel server. The channel server stores the <hostname, port, subscriptionType> information in a lookup table, grouped by channel names. It also returns information about all the other <hostname, port> pairs subscribed as sink/source to this channel. Once the peer information is received, dQUOBEC establishes a point to point connection between the current tasks and all its peers. The data is transferred as structured events. Additionally the sink task needs to specify a handler function that gets executed at the receipt of every event. The task can now send/receive information through these connections depending on whether it is a source or a sink.

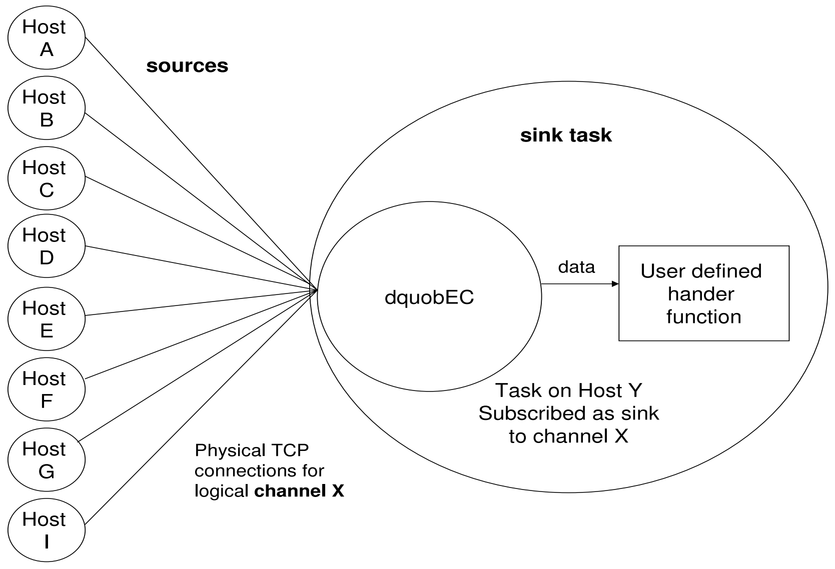
Host A
Host B
Host C
Host D
Host E
Host F
Host G
Host I
sources
sink task
dquobEC
data
User defined hander function
Task on Host Y Subscribed as sink to channel X
Physical TCP connections for logical channel X

Figure 2. Data handling at the sink task

dQUOBEC is implemented as a library that is linked into the source or sink task as shown in Figure 2. The library at the sink, for instance, manages multiple channels by creating a dedicated poller thread for each channel that has been subscribed to. This receiver thread polls all connections of a channel for input data. dQUOBEC also establishes a common threadpool for handling data received from all channels. When the channel specific poller thread receives data, it copies the data into the common threadpool queue. These data are then handed over to their corresponding handlers by the threads in the threadpool. This design decouples receiving of data from its handling and enables high scalability. dQUOBEC depends on the underlying TCP layer to take care of delays in connection request/response and data loss due to network problems.

5. Experimental Analysis

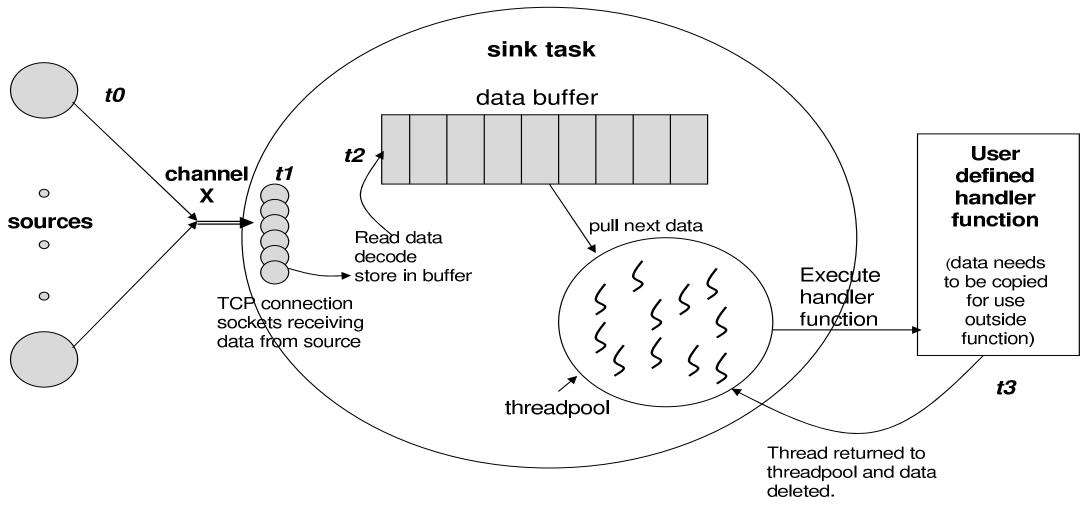
sink task
t0
data buffer
t2
channel X
t1
sources
User defined handler function
(data needs to be copied for use outside function)
pull next data
Read data decode store in buffer
Execute handler function
TCP connection sockets receiving data from source
threadpool
t3
Thread returned to threadpool and data deleted.

Figure 3. Timing analysis for dQUOBEC.

This section describes the experimental analysis of dQUOBEC. The analysis is conducted in two parts. We measure the *end-to-end* latency time and *data transfer* time between an arbitrary source task and a known sink task in the presence of multiple other source tasks, sink tasks and channels. The experimental setup employs a cluster of Solaris and Linux machines connected through 1Gbps ethernet LAN for conducting the above tests. Table 1 provides details of the machines that run the measured parts of the experiment, that is the source and sink.

End-to-end latency time is the time taken for the source task to issue a send request using dQUOBEC and the sink task to receive the corresponding data in its receiving socket. The receiving socket is internal to dQUOBEC and is not visible to the sink task. This experiment measures the overhead of dQUOBEC on top of TCP/IP. This is shown as $(t_2 - t_0)$ in Figure 3. The end-to-end latency time includes the time taken to transfer the data event through the socket

Host: tamarack.cs.indiana.edu (source)	Host: bandtail.cs.indiana.edu (sink)
Sun Blade 1000 Model 1900	Sun Blade 100 Model 500
CPU: 900 MHz sparcv9+vis2 CPU	CPU: 502 MHz sparcv9+vis CPU
OS: Solaris 8	OS: Solaris 8
Kernel: 5.8 Generic(64 bit)	Kernel: 5.8 Generic(64 bit)
Memory: 1.0 GB real memory	Memory: 512 MB real memory

Table 1. Machine Specification for the experimental analysis.

No. of Channels	No. of Sources	No. of Sinks	No. of records at the channel server	Decode time $(t_2 - t_1)$ (msec)	End to End latency $(t_2 - t_0)$ (msec)
1	1000	1	1001	0.5	100
1	1	1000	1001	0.5	1
10	10	10	400	0.5	15
10	95	95	1900	0.5	10
100	5	5	1000	0.5	20
500	1	1	1000	0.5	1

Table 2. End-to-end latency time $(t_2 - t_0)$.

$(t_1 - t_0)$, plus the time taken to decode the data $(t_2 - t_1)$. The decode time $(t_2 - t_1)$ varies with the size of data.

Data transfer time is the end-to-end latency time plus the time taken to execute the handler function. The handler function is executed by a dQUOBEC thread for each event received through the channel. The data transfer time is measured as $(t_3 - t_0)$ in our experiment shown in Figure 3.

Experiment 1: The first experiment uses events of a small size (50 KB) and generation rates of 1 event/sec. This experiment verifies the ability of dQUOBEC to handle multiple sources and sinks for a large number of channels at the same time. It also measures the end to end latency, that is the time duration between the sending of an event by a source and its receipt by the sink. We measured the end-to-end latency time $(t_2 - t_0)$, in Figure 3 between an arbitrary source task and a known sink task subscribed to a specific channel, in the presence of multiple other channels, source tasks and sink tasks. We measured the elapsed time from the instant the source sends data to the instant it is received by the sink. The results are shown in Table 2. The end-to-end latency between an arbitrary source and the receiving sink consists of the time taken to encode the data into a binary format, transfer the binary data to the receiving task and decode the data from binary format to a given format. The values provided in table 2 has been averaged over many runs.

Note: Each subscription to a channel needs one socket and every established connection uses up one socket descriptor. Our test environment supported a maximum of 1024 file/socket descriptors per process. Hence the tests were constrained by the maximum number of sockets in a process that could be open at the same time. Measurements were taken after adjusting for clock skews.

Experiment 2: The second experiment uses large events (3 MB), and a fan-in arrangement of some number of source

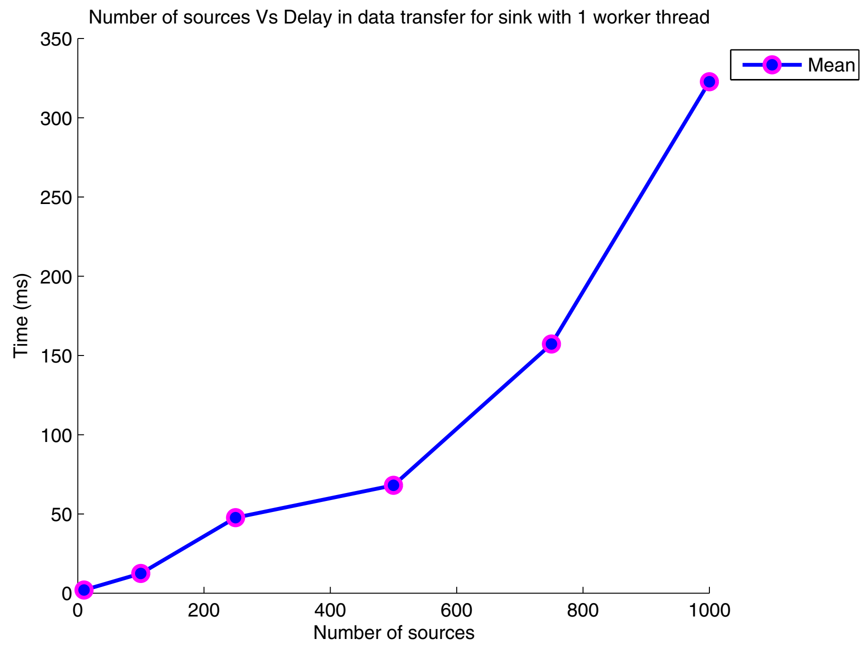

Figure 4. Increase in data transfer time ($t_3 - t_0$) of figure 3 due the presence of other source tasks sending 3 MB data at the same time. The sink task started with 1 worker thread. The time taken to execute the handler function for each event is approximately 1 second.

tasks sending events to the same sink task. Each source generates data at the rate of 1 event/sec. It measures *data transfer time*, that is the time taken to transmit the data from a source task to the sink task and execute its corresponding handler function. We show how the time taken for executing the handler function influences the data transfer time of dQUOBEC. *If the data is needed outside the handler function, it needs to be copied from dQUOBEC's buffer into the application buffer by the handler function.* dQUOBEC de-allocates memory used to store events, after executing the handler function.

We measure the time taken to transfer a large data event from a source to a sink through dQUOBEC, under varying conditions. This is shown in Figure 4. The time taken for data transfer increases when there are other sources in the system sending data through the channel at the same time. The data transfer time is influenced by the time taken to read the data from the socket into the buffer and execute the handler function for each data event. The number of worker threads in thread pool of the sink process is another parameter that affects the data transfer time. The receiver thread reads events into to the thread pool buffer queue. The worker threads in the thread pool are responsible for executing the corresponding handler functions for the data in its queue.

The sink was started with one worker thread for thread pool. Since the data transfer time varies with the order in which data is received, utmost care was taken to ensure that all sources send data at the same time. Thus, the actual order in which data was received is quite random for all runs. The measured values were averaged over many runs to remove any bias towards the order in which connections were established.

Analysis: Analysis of Table 2 shows that dQUOBEC is capable of simultaneously supporting multiple sources and sinks for a single channel. The time taken for encoding and decoding are low because the event size is small (50KB). Note that only non binary data is encoded at the sender and decoded at the receiver. If an application transmits a binary data using dQUOBEC there will be no format conversions. From Table 2, we can see that the low overhead of dQUOBEC advocates for its high scalability in supporting a large number of individual channels.

Figure 4 shows the increase in data transfer time to transfer 3 MB of data from a single source to a sink in the presence of 10, 100, 250, 500, 750 and 1000 other sources on different machines, all sending data at the same time. When multiple sources send data simultaneously, the effect is that events can arrive at the sink in any order. This randomness in arrival "order" impacts the latency of source to sink measurements because of the arbitrary amount of time an event may await handling at the sink. Varying the number of worker threads in the thread pool of the sink changes the time taken to invoke handlers for incoming data. By using multiple threads in the common threadpool, the data transfer time can be reduced. The analysis of Figure 4 shows that dQUOBEC is capable of handling data from multiple sources of varying sizes simultaneously with low data transfer time.

6. Conclusion and Future Work

In this paper, we discussed dQUOB's motivating requirements that led to the development of dQUOBEC. dQUOBEC is a subject based publish-subscribe system with a push based streaming model and a peer-to-peer architecture. The overhead of dQUOBEC is very low as seen from the experimental analysis. We conclude by stating that dQUOBEC is a light weight and efficient system capable of transferring large binary data. dQUOBEC can be extended to include the following features:

Data rate for a channel: Assuming the rate at which data arrives in a channel is the same as the rate at which data is delivered (no loss), this rate can be calculated either at the sending end or the receiving end. If it is done at the sending end, then all sources need to share this information through a common entity that calculates the data rate of a channel. We suggest calculating data rate at the receiving side as the sink receives all events sent on a channel provided there are no losses in the network. This information can be communicated to others by storing it in dQUOBEC's channel server.

Subscription renewal: The channel server needs to notified if a process completes or a host goes down. This feature can be implemented by using a lease renewal technique where all source tasks and sink tasks need to renew their subscriptions at regular intervals of time. If a task has completed, it can unsubscribe from a channel or will be automatically removed if it does not renew its lease.

Support for other binary formats: Currently dQUOBEC supports PBIO data format. Work is in progress to support HDF5 [5] and netCDF [6] formats.

References

[1] Beth Plale and Karsten Schwan. Dynamic Querying of Streaming Data with the dQUOB System. *IEEE Transactions on Parallel and Distributed Systems, Vol. 14, Number 3*, 2003.

[2] Fabian Bustamente, Greg Eisenhauer, Karsten Schwan and Patrick Widener. Efficient Wire Formats for High Performance Computing. In *Proceedings of Internaltional Conference for High Performance Computing Networking and Storage (Supercomputin)*, 2000.

[3] G. Fox and S. Pallickara. An event service to support grid computational environments. *Journal of Concurrency and Computation: Practice and Experience. Special Issue on Grid Computing Environments*, 2002.

[4] Greg Eisenhauer and Fabian Bustamente and Karsten Schwan. Event services for high performance computing. In *Proceedings of 9th IEEE Intl. High Performance Distributed Computing (HPDC)*, 2000.

[5] Hierarchical Data Format 5. HDF5. http://hdf.ncsa.uiuc.edu/HDF5.

[6] network Common Data Form. Unidata-NetCDF. http://my.unidata.ucar.edu/content/software/netcdf/index.html.

[7] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of Networked Group Communication*, pages 30 – 43, 2001.

[8] A. Slominski, Y. Simmhan, A. Rossi, M. Farrellee, and D. Gannon. Xevents/xmessages: Application events and messaging framework for grid. Indiana University, Extreme Lab, 2001.

[9] Ying Liu and Beth Plale, Indiana University. Survey of Publish Subscribe Event Systems. Technical report, Indiana University, IU-CS TR574, 2003.

A. Appendix

A.1. API

Channel_element();

    -Constructor for channel element.

 Channel_element();

    -Destructor for channel element.

int init();

    -Initialize the channel element. Check for return value.

int subscribe_source(char *name, IOFieldList io);

-Subscribe as a source to a channel with channel name and data format.

int subscribe_sink(char *name,
    void (*handler) (void * event, void * client_data),
    void * client_data, IOFieldList io);
    -Subscribe as a sink to a channel with channel name, pointer to handler
function, pointer to client data and data format.

int send(char *name, void *event, long size);
    -Send event of given size through channel

int update_client_data(char *name, void *client_data);
    -Update client data for a sink subscription.

int update_handler (char *name, void (*handler)(void *event,
    void *client_data));
    -Update handler function for a sink subscription.

int unsubscribe_source(char *name);
    -Unsubscribe from being a source to a channel.

int unsubscribe_sink(char *name);
    -Unsubscribe from being a sink to a channel.

A.2. Example

This section demonstrates use of dQUOBEC by source and sink processes for a channel. In the example the channel is used for transferring virtual memory statistics.

Code for the source process uses Solaris kstat library to get virtual memory information and sends this information as events in the channel vm_channel using dQUOBEC. The sink process receives the virtual memory information of all sources through the vm_channel. Upon receiving the events, a function to display incoming virtual memory information is invoked in the sink.

protocol.h: A sample structure of data being transmitted is defined here

```
/*****************
protocol.h
*****************/

#ifndef PROTOCOL_H
#define PROTOCOL_H

/*Virtual Memory Statistics structure declaration*/
typedef struct VMstats_rec_{
  char *hostname;
  long freemem;
  long swap_resv;
  long swap_alloc;
  long swap_avail;
  long swap_free;
}VMstats_rec, *VMstats_recPtr;

#endif
```

formats.h: The corresponding pbio format for data being transmitted is defined here

```
/*****************
formats.h
*****************/

#include "io.h"
#include "protocol.h"

#ifndef FORMATS_H_
#define FORMATS_H_

/*Virtual Memory Statistics PBIO declaration*/
static IOField VMstats_rec_IO_list[]={
  {"hostname", "string", sizeof(char*), IOOffset(VMstats_recPtr, hostname)},
  {"freemem", "integer", sizeof(long), IOOffset(VMstats_recPtr, freemem)},
  {"swap_resv", "integer", sizeof(long), IOOffset(VMstats_recPtr, swap_resv)},
  {"swap_alloc", "integer", sizeof(long), IOOffset(VMstats_recPtr,
                                                   swap_alloc)},
  {"swap_avail", "integer", sizeof(long), IOOffset(VMstats_recPtr,
                                                   swap_avail)},
  {"swap_free", "integer", sizeof(long), IOOffset(VMstats_recPtr, swap_free)},
  {NULL, NULL, 0, 0}
};

#endif
```

source.C: An example provider / publisher

```
/*****************
source.C
*****************/

#include <kstat.h>
#include "Channel_element.H"
#include "formats.h"
#include <sys/sysinfo.h>

#define BYTES_PER_PAGE 8192
#define KBYTE            1024
#define MUL              (BYTES_PER_PAGE/KBYTE)
#define MAXHOSTNAMELEN 256

int main(int argc, char*argv[]){

  kstat_ctl_t    *kc;
  kstat_t        *ksp;
  int            first =1;
  vminfo_t  vm,last;

  /*Instantiate channel element and assign channel*/
  Channel_element ce;
  ce.init();
  char *vmchannel = "vm_channel";
  VMstats_rec vmstats_rec;
  char *hostname = new char[MAXHOSTNAMELEN];

  gethostname(hostname, (size_t)MAXHOSTNAMELEN);

  /*Subscribe as source for vmchannel*/
  if(ce.subscribe_source(vmchannel, VMstats_rec_IO_list)<0){
    cout<<"Subscription as source for channel "<<vmchannel<<" failed"<<endl;
    exit(1);
  }

  kc  = kstat_open();
  while(1) {
    /*Make a kstat call to get virtual memory info*/
    ksp = kstat_lookup(kc, "unix", 0, "vminfo");

    if (ksp) {
      kstat_read(kc, ksp, &vm);

      if (first) {
        first = 0;
      }
      else {
        vmstats_rec.hostname = hostname;
        vmstats_rec.freemem = (vm.freemem−last.freemem) * MUL;
        vmstats_rec.swap_resv = (vm.swap_resv−last.swap_resv) * MUL;
        vmstats_rec.swap_alloc = (vm.swap_alloc−last.swap_alloc) * MUL;
        vmstats_rec.swap_avail = (vm.swap_avail−last.swap_avail) * MUL;
        vmstats_rec.swap_free = (vm.swap_free−last.swap_free) * MUL;
        ce.send(vmchannel, (void *)&vmstats_rec, (long)
                sizeof(vmstats_rec));
      }
      last = vm;
    }

    /*Wait for 1ms and repeat*/
    usleep(1000000);
    kstat_chain_update(kc);
  }
  kstat_close(kc);
  return(0);
}
```

sink.C: An example receiver / subscriber

```
/*****************
sink.C
*****************/

#include "Channel_element.H"
#include "formats.h"

/*Declare function to be invoked when event arrives*/
void print_vm_data(void *,void *);
```

```
int main(){

  function_handler fp1 = &print_vm_data;
  char *vmchannel = "vm_channel";
  Channel_element ce;
  ce.init();
  int value;


  /*Subscribe as sink to vmchannel*/
  if(ce.subscribe_sink(vmchannel, fp1, NULL, VMstats_rec_IO_list)<0){
    cout<<"Subscription as sink for channel data_ev failed"<<endl;
    exit(1);
  }

  cin>>value;
  return 0;
}

/*Function to be invoked when data is received*/
void print_vm_data(void *p,void *q){
  VMstats_rec *vmstats_rec=(VMstats_rec *)p;
  printf("\n----------------------------------\n");
  cout<<"VM stats"<<endl;
  printf("freemem        =%ld KByte\n",vmstats_rec->freemem);
  printf("reserved swap =%ld KByte\n",vmstats_rec->swap_resv);
  printf("allocated swap=%ld KByte\n",vmstats_rec->swap_alloc);
  printf("avail swap    =%ld KByte\n",vmstats_rec->swap_avail);
  printf("free swap     =%ld KByte\n",vmstats_rec->swap_free);
  printf("----------------------------------\n");
}
```