

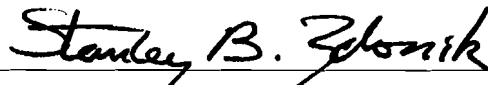
# The Pub-Sub Simulator

Rahul Bose

Department of Computer Science  
Brown University

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science  
in the Department of Computer Science at Brown University

January 1999

A handwritten signature in black ink that reads "Stanley B. Zdonik". The signature is written in a cursive style with a horizontal line underneath it.

Dr. Stanley B. Zdonik

# Contents

<b>Acknowledgments</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Publish-Subscribe Systems . . . . .	1
1.2 The Pub-Sub Simulator . . . . .	1
<b>2 Simulator Design</b>	<b>3</b>
2.1 The Publish Subscribe Model . . . . .	3
2.2 Design Overview . . . . .	3
2.3 Modules . . . . .	4
2.3.1 The Server . . . . .	4
2.3.2 The Profiler . . . . .	4
2.3.3 The Mapper . . . . .	4
2.3.4 The Scheduler . . . . .	4
2.3.5 The Client . . . . .	5
2.3.6 The ClientManager . . . . .	5
2.3.7 The ChannelManager . . . . .	5
<b>3 Metrics</b>	<b>7</b>
3.1 Types of Measurement . . . . .	7
3.2 Staleness . . . . .	7
3.3 Queue Lengths . . . . .	8
3.4 Differential Flow Rate . . . . .	8
3.5 Update Time At Liftoff . . . . .	9
<b>4 Channel Assignment</b>	<b>11</b>
4.1 Client-To-Channel Mapping . . . . .	11
4.2 Algorithms . . . . .	11
4.2.1 Random . . . . .	11
4.2.2 RR . . . . .	12
4.2.3 PC . . . . .	12
4.2.4 LC . . . . .	12
<b>5 Mapping Algorithms</b>	<b>15</b>
5.1 Page-to-Channel Mapping . . . . .	15
5.2 Algorithms . . . . .	15
5.2.1 Random . . . . .	15
5.2.2 C . . . . .	15

# Acknowledgments

The work described in this report is the result of almost a year's work by the members of the Data Dissemination Group – myself, Donald Carney, and Kerry Kurian, with guidance from Professor Stanley Zdonik.

For purposes of obtaining credit towards a masters degree in Computer Science, the following describes my specific contributions to the project.

- The page-to-channel mapping algorithms CD, Offline\_C, and Perfect\_C
- The client-to-channel assignment algorithms RR and LC
- The classes ChannelManager, ClientManager, Profile, ZClusProf, CM\_RR, CM\_PC, TList

I have also been involved from the beginning in the evolving design of the Pub-Sub Simulator, and have played a leading role in maintaining the code base.

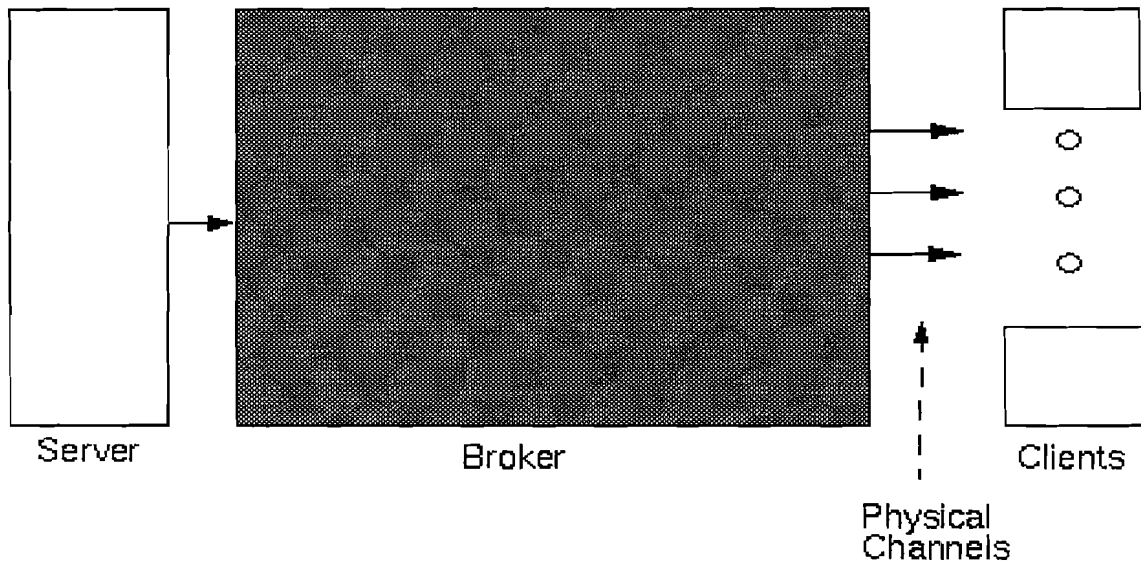
I wish to thank the other members of the Data Dissemination group - Donald Carney, and Kerry Kurian. I am grateful to my advisor Prof. Stanley Zdonik for his guidance during my stints first as a student, and then as a research staff member.

# Chapter 1

## Introduction

### 1.1 Publish-Subscribe Systems

A publish subscribe system consists of a number of data sources or publishers, a very large number of clients or subscribers, and one or more agents in between. These agents are responsible for figuring out what data should reach which clients, and how it should be routed. The clients make their interests known by submitting interest profiles to the agent.



### 1.2 The Pub-Sub Simulator

The Pub-Sub simulator attempts to study the issues involved in building such publish subscribe systems. In our model, the publish subscribe agent gets data from the data sources and transmits them to the clients through a set of broadcast channels to which the clients are listening. There are typically many more clients than there are channels, and each client has a limit on the maximum number of channels they can listen to.

# Chapter 2

## Simulator Design

### 2.1 The Publish Subscribe Model

The model we use of a publish subscribe system consists of data sources on one end which keeps updating data, clients on the other end who submit profiles listing their interests, and an agent or broker in the middle which routes and schedules the data.

In our current implementation, we also introduce the following criteria:

- Every client can listen to a certain maximum number of channels, which is less than the total number of channels available in the system.
- All clients have equal importance, i.e. we have not yet introduced client priorities.
- Client profiles do not have priorities linked to them, i.e. all pages are of equal importance to the client. However, there can exist hot-spots in the client interests, i. e. pages in which large numbers of clients are interested.
- Data pages can be of different sizes. The amount of time taken to transmit a page is directly proportional to its size.

### 2.2 Design Overview

There are five main modules in the publish subscribe simulator which model different aspects of the system. These are:

**Server** The server generates random page updates and sends them to the profiler.

**Profiler** The profiler matches page updates to client profiles and lets those pages which are of interest, through into the system.

**Mapper** The mapper determines how to route each page so that every interested client receives it.

**Scheduler** The Schedulers (one for each channel) decide in which order to deliver the pages to the clients.

**Client** The clients listen on the channels and receive pages of interest.

the page. The RxW algorithm also takes into account the time a page has spent in the buffer, in order to prevent starvation.

Each of the schedulers act independently, and once a page has been put into a channel, it will always eventually reach the clients who are listening on the other end. One of the areas of future work is a scheduler which drops pages selectively as the load on the channel increases.

### 2.3.5 The Client

The Client object acts on behalf of all the simulated clients in the system. It listens to all the channels, and keeps measurements of pages received on a per client basis.

### 2.3.6 The ClientManager

The ClientManager generates and stores information about all the clients in the system. It acts as a central repository of client data which other objects in the system can query. Examples of the kind of queries possible are :

- Which channels is a given client listening to?
- Which clients are interested in a certain page?
- What are the pages a specified client listening to?
- Which clients are listening to a given channel?

The ClientManager maintains several data structures which contain all of this information. It also helps generate this information with the help of other objects such as the ChannelManager (which determines client-to-channel mappings) and the Profile object (which generate client profiles).

### 2.3.7 The ChannelManager

The ChannelManager determines which clients will listen to which channels. For this, it takes into account the client interests, as well as the server page generation probabilities.

The ChannelManager is one of the more important modules in the simulation, since the client-to-channel mapping determines to a large extent the performance of the entire system. Details of the various algorithms and their relative performances can be found in the chapter on Channel Assignment.

# Chapter 3

## Metrics

### 3.1 Types of Measurement

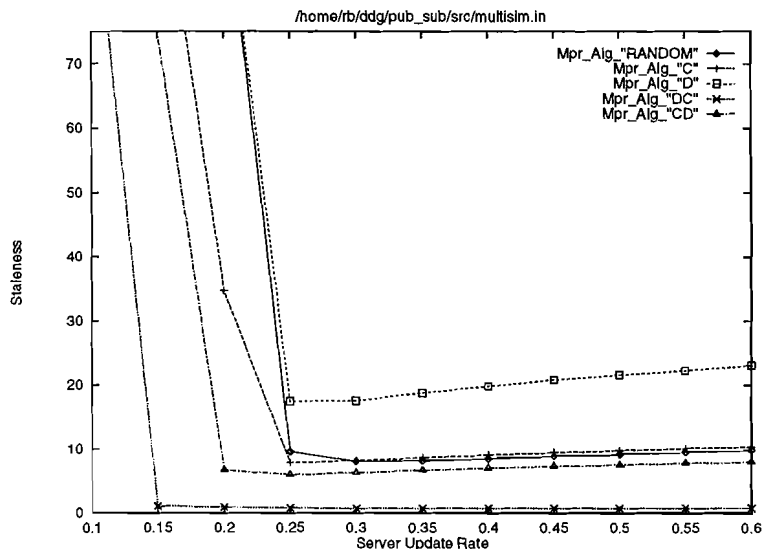
The development of metrics which accurately portray the performance of the system was one of the more significant tasks faced during the design of the simulator. The current metrics which are built into the simulator approach this problem in two ways. One set of metrics measure the system from the point of view of a client. The other set of metrics look at different parts inside the system and measure quantities such as rate of flow, etc.

### 3.2 Staleness

We define **Staleness** of a page as the time difference between the update time of the page at the server, and the receive time of the page at the client. In our simulation, this is also equal to the time the page spends inside the system, since a page is pushed out from the server immediately after updation.

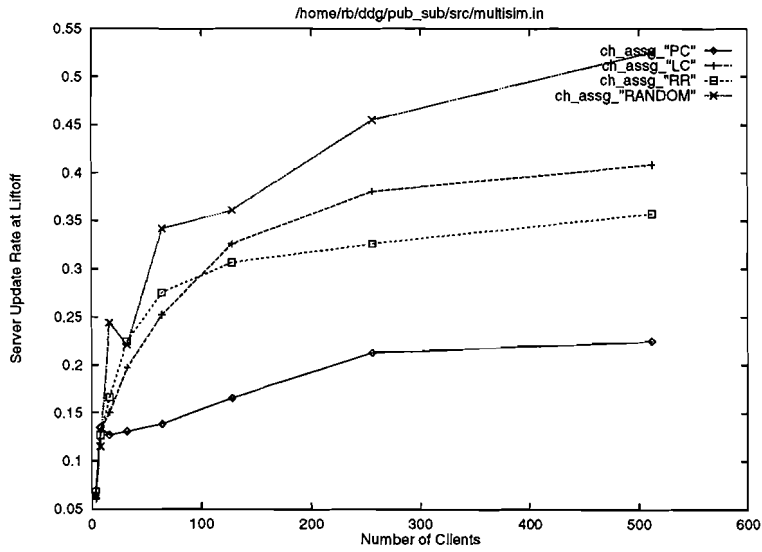
The **Average Staleness** measured at a client is the average of the staleness of all the pages received at that client over the course of the experiment.

The **Global Staleness** is the mean of all the **Average Staleness**'s measured at each of the clients.



### 3.5 Update Time At Liftoff

The Update Time At Liftoff (UTAL) shows the server update rate at which the DFR of the system passes a certain small threshold. In other words, for a given system, as we ratchet up the update rate at the server, the UTAL shows the point at which the system starts getting overloaded.





# Chapter 4

## Channel Assignment

### 4.1 Client-To-Channel Mapping

The problem of deciding which clients should listen to what channels is a non-trivial one, as one has to take into account client interests and page update rates at the server, while attempting to achieve the following goals:

- Reducing the amount of *fluff* a client has to receive, i.e. data which is not of interest to the client.
- Distributing load across the channels uniformly.
- Enabling the Mapper to reach all interested clients using the minimum number of channels for each data item.

Reducing *fluff* is achieved by grouping or clustering clients with similar interests together on channels. This leads to less time wasted by clients on filtering out unwanted data. Clustering similar clients together also helps the mapper reach all interested clients using fewer channels.

A channel allocation algorithm also needs to take care that it does not load up any one channel much more than others, since this channel then becomes the weak point in the system and the first to break as the data load goes up. Different algorithms take different approaches to balancing load. The simpler ones estimate channel load by the number of clients listening on the channel. More complex algorithms look at the page generation probabilities of the pages which will be going through that channel.

Estimating channel load is complicated by the fact that a lot depends on the the page mapping algorithm being used at the Mapper. And conversely, the performance of a page mapping algorithm depends on the type of channel allocation strategy used.

### 4.2 Algorithms

#### 4.2.1 Random

This simple algorithm randomly allocates channels to clients. For large client populations, this results in a uniform distribution of clients among channels. It performs especially well when all the client profiles are alike, since channel load then gets uniformly distributed.

For each channel

    Randomly allocate a client to that channel

Set chID to 1

While there exists eligible clients

    Pick a client whose profile best matches the profiles of clients already on channel chID

    Assign this client to channel chID

    If the client's channel quota is filled

        Remove the client from list of eligible clients

    Increment chID by 1. Wrap around if it exceeds total number of channels

Note: To pick the best client for a given channel, the number of pages common between the client's profile, and the combined profile of clients already listening to the channel, is considered. Pages having a higher P value (i.e. greater server generation probability) are given higher weight. Ties are broken randomly. The term LC stands for Load-balanced Clustering.

# Chapter 5

## Mapping Algorithms

### 5.1 Page-to-Channel Mapping

It is the Mapper's responsibility to ensure that each updated page generated by the server reaches all the clients interested in it. For this task, it implements a number of mapping algorithms which determine which page should be sent out on which channels. These algorithms can be roughly divided into two groups: online algorithms – where the page-to-channel mappings are static, and offline – where they are determined on the fly.

Each channel has an associated maximum throughput. In order to load the system as little as possible, an efficient mapper attempts to send the page out on the minimum number of channels possible. At the same time, it also tries to balance load across all channels.

### 5.2 Algorithms

#### 5.2.1 Random

This is a very simple algorithm which serves as a benchmark for the others. When a page comes into the mapper, the set of outgoing channels is picked in the following manner.

While there exist interested clients

- Randomly pick a channel

- If any interested clients are listening on the channel

  - Send a copy of the page out on that channel

  - Remove these clients from the interested clients list

#### 5.2.2 C

This algorithm tries to minimize the number of channels a page has to be sent out on. To determine this, it uses the following greedy algorithm.

While there exist interested clients

- Pick the channel which has the most number of interested clients

- In case of a tie, randomly pick one of the tied channels

  - Send a copy of the page out on that channel

  - Remove these clients from the interested clients list

# Appendix A

## Class Definitions

### A.1 The Simulation Class

The Simulation class implements the main Simulation object which runs and controls the entire simulation. It accepts the simulation parameters from the user, starts the various modules, decides when to end the experiment, and reports results back to the user.

#### Operations

**start** Runs the simulation. Returns after the simulation is done.

#### Private Functions

**CreateChannels** Constructs the various channel objects

**CreateModules** Constructs the main modules of the simulation

**InitModules** Calls the Init function of each module, passing in pointers to other modules

**StartModules** Calls the Start function of each module

**ReadLiftoffInput** Deciphers the input parameters for the liftoff code

**FindUpdateTimeAtLiftoff** Implements the code for finding the liftoff point

#### Notes

The Simulation start function is called by main(). It begins by reading input parameters from the run.in file, and then constructs all the major modules, passing each of them their corresponding input parameters.

In experiments without liftoff measurements, the simulation then starts off all these modules, waits for a user specified amount of time, shuts down the server and returns. Control flows back to main(), which obtains the required metric from the Simulation object, outputs it and exits.

By the term *liftoff*, we are referring to the fact that when we plot the GDFR metric as a function of server speed, we observe that as speed increases, the GDFR remains fairly constant, and then suddenly lifts off at a sharp slope. The liftoff code tries to automatically

## A.2 The Server Class

The Server class implements the Server object in the Simulator. It contains one or more data sources, which generate page updates according to some random distribution. These updates are then sent to the Profiler object.

### Initialization Parameters

**int server\_type** The page distribution algorithm. This can be Uniform, Zipfian, or Poisson

**int num\_data\_sources** Equivalent to number of hotspots in the distribution

**int num\_items\_per\_data\_source** num pages in each hotspot

**data\_element\_size** The size of each page generated. Used to simulate page transmission time

**mean\_update\_time** Mean time between updates at server. An exponential distribution is used around this mean time.

**zipf\_desc** Passed to the Zipf object if the server\_type is Zipfian

### Operations

**Start** Start sending pages

**Stop** Stop sending pages

**GetPageGenerationProbs** Return a list giving the generation probability of each page. First entry is for page 1, the last for page `dBaseSize`

**GetDBaseSize** Return the total number of pages in the server database. i. e. `num_data_Sources` X `num_items_per_data_source`

**GetNumSources** Return number of hotspots in the database, i. e. `num_data_Sources`

**SetMeanUpdateTime** Set the mean time between page updates. i. e. `tt mean_update_time`. Used by the liftoff code to change the server speed

**OriginalMeanUpdateTime** Get the mean update time which had been passed into the constructor. This is used by the liftoff code if it decides to run the original experiment.

### Protected Functions

**ReadInput** Deciphers the input description passed into the constructor

### Notes

## A.3 The Profiler Class

The Profiler class implements the Profiler object in the Simulator. It is placed in front of the server and acts as a filter by letting through only those pages which are of interest to one or more clients in the system. It also helps in taking certain measurements.

### Operations

**Init**     Receives pointers to the ClientManager and Client objects

**Start**     Gets into a loop waiting for pages from the server

**GetFlowRate**    Get rate at which pages are flowing out of the Profiler

**ResetFlowRate**   Reset the flow rate to zero. Called by the Simulation liftoff code in preparation for running a new experiment

**MeasureGlobalStats**   Set the measurement flag “\_global\_stats”. Enables the measurement of flow rate.

**MeasureLocalStats**   Reset the measurement flag “\_global\_stats”. Disables the measurement of flow rate.

### Private Functions

**GetMessages**    Gets a message from the server. Decides whether to pass it on.

**SendMessages**   Sends the message to the Mapper. Updates flow rate

### Notes

The Profiler is one the simpler modules in the simulator. It consists of a CSim process which keeps waiting for pages from the server, asks the ClientManager whether a received page is of interest to any client, and if so, sends the page to the Mapper. If no clients are interested, it simply drops the page.

To help in SDFR measurements, the profiler tells the Client about each page it sends. The Client in turn uses this information to calculate the percentage of pages it has successfully received.

The profiler also keeps track of the rate at which it is passing pages on to the mapper. This measure is not used in any metric at the moment, but can be obtained as output by specifying the output\_type to be PROFILER\_FLOW\_RATE. Note that the MeasureGlobalStats function will have to be called first to enable measurement of flow rate.

the way the server is set up, this also equals the server update rate at which the channel will be operating at full capacity. Thus for offline algorithms, by looking at the TotalP values of the channels, we can predict the server speed at which each channel breaks. An example of this table is given below. In this example, channel 0 would be the first to break as we increase server speed to an update rate of 0.202. i.e. mean time between updates at server = 0.202 time units.

```
Mapper::AnalyzeMapping()  
Channel 0: NumPages = 2, TotalP = 0.202219  
Channel 1: NumPages = 2, TotalP = 0.103672  
Channel 2: NumPages = 5, TotalP = 0.103469  
Channel 3: NumPages = 5, TotalP = 0.109922  
Channel 4: NumPages = 4, TotalP = 0.0579375  
Channel 5: NumPages = 3, TotalP = 0.0712656  
Channel 6: NumPages = 3, TotalP = 0.0645312  
Channel 7: NumPages = 5, TotalP = 0.0725781
```

The  $R \times W$  value of a page is the product of its  $R$  and  $W$  values, where the  $R$  value is the number of clients interested in the page, and the  $W$  value is the number of time units the page has been in the buffer. Essentially, the  $R$  value gives a higher priority to more important pages, while the  $W$  value prevents starvation of the less important pages.

The Scheduler keeps track of the rate at which pages are flowing in and out of the channels. The GDFR metric is calculated from these as `input_flow_rate - output_flow_rate`, totalled across all schedulers.



### Notes

The Client object listens to the channels on behalf of all the clients in the system. Since the client profiles are generated and maintained by the ClientManager, the Client is a lightweight object which only takes measurements. Two of these measurements are global staleness, and SDFR.

Three functions are used for measuring SDFR. The first is **SendingMessage**, which gets called by the Profiler each time it sends a new page out. The function records the packet number and the list of clients interested in an internal table. The second function is **ReceivedMessage**, which marks off the clients who have received the packet. Once all clients have received the packet, it is removed from the internal table. The third function is **GetSDFR**, which returns the number of packets remaining in the internal table, divided by the total experiment time. Thus SDFR measures the rate at which packets get backed up in the channels. A packet is said to be backed up as long as there exist some interested clients who have not yet received it.

The ClientManager first looks at its input parameters and decides which Profile and ChannelManager objects are to be used.

It then generates and stores all the client profiles with the help of the Profile object. Next, it calls upon the ChannelManager object to generate the client-to-channel mappings.

The ClientManager provides a large number of accessor functions for access to all this data. Queries about profiles are answered directly, while queries about the channel assignments are redirected to the ChannelManager.

One of the later modifications to the ClientManager involve generating a fixed channel assignment across multiple experiments. For this, it first generates the channel allocation using one set of client profiles (*BuildStartupGlobalProfiles*), then throws away the profiles and generates fresh ones using a different distribution (*BuildRuntimeGlobalProfile*). Since the channel allocations depend on the client profiles, we can fix the channel assignment across experiments by using the same startup profiles.

## A.9 The ZClusProf Class

The ZClusProf class generates a zipfian, clustered client profile. It is derived from base class Profile.

### Initialization Parameters

- int numClusters**    number of hotspots in profile. Each hotspot will correspond to a source in the server
- int numHotSpots**    number of hotspots, i.e. sources, at the server. These are equispaced in the database range
- int clusterSize**    number of pages in each cluster
- int clusterRange**    max pageID for a cluster. This is equivalent to num pages per source at the server

### Operations

- Generate**    Generates a client profile, i.e. a list of pages which is a subset of the entire database

### Notes

A client profile consists of a list of page numbers. The ZClusProf generates randomly generates such a list of pages using one or more zipfian distributions.

In a zipfian distribution, a relatively small number of pages are very hot, i.e. they have a high probability of appearing. As we move away from this hotspot, the pages rapidly become colder. The exact probability curve is defined by certain parameters which are supplied to the internal Zipf object used by the Zipfian class.

In our current experiments, client profiles have one hotspot. Page number 1 is the hottest page, and page number *dBaseSize* is the coldest.

The Zipfian class also has the capability to generate a profile containing several hotspots or clusters. The *numClusters* parameter specifies the number of clusters in a client profile. The hotspot of each of these clusters must coincide with a hotspot at the server. Since hotspots at the server divide the *dBaseSize* page range equally, it is enough to specify the number of hotspots at the server, for the client to figure out the location of the hotspots. The *numHotspots* parameter gives this information.

Note that *numClusters* must always be less than or equal to *numHotspots*. The current clustering model is designed to reflect a scenario where a number of sources publish at the server, and a client subscribes to one or more of these sources.

Also note that multiple clusters do not make sense with a uniform server or uniform client profile. Uniform profiles are generated by the Profile class, which ignores the *numClusters* input parameter.

## A.11 The CM\_Random Class

The CM\_Random class is derived from the base class ChannelManager. It generates a client-to-channel mapping based on a random distribution.

### Initialization Parameters

**int channelsPerClient** Max number of channels each client can listen to

**int totChannels** Number of channels in the system

**int numClients** Number of simulated clients in the system

### Operations

**AllocateChannels** Implements the random allocation algorithm

### Private Functions

**AllocateChannels** Allocates channels for a given client and calls AddChannelInfo

**AddChannelInfo** Adds the channel list to internal data structure which maps a client to channels, and then calls AddClientInfo

**AddClientInfo** Adds a client to internal data structure which maps a channel to clients

### Notes

For a description of the Random algorithm, refer to the chapter on Channel Allocation.

The thread of control through the various functions runs as follows:

The main *AllocateChannels* function goes through the list of clients, and for each client calls another function, also named *AllocateChannels* which allocates channels for that particular client. This function determines the set of channels and calls *AddChannelInfo*, which in turn calls *AddClientInfo*.

## A.13 The CM\_PC Class

The CM\_PC class is derived from the base class ChannelManager. It generates a client-to-channel mapping based on a load balancing algorithm which gives priority to hot pages.

### Initialization Parameters

**int channelsPerClient** Max number of channels each client can listen to

**int totChannels** Number of channels in the system

**int numClients** Number of clients in the system

### Operations

**AllocateChannels** Implements the round robin channel allocation algorithm

### Private Functions

**PickPageWithMaxPC** Returns page with the max PC value.

**AssignChannelToClients** Modifies internal data structure to assign a channel to given clients.

**AssignPageToChannel** Temporarily assign a page to a channel.

**GetPageList\_Channel** Gets the list of pages assigned to a channel.

**GetLeastLoadedChannel** Gets the channel having the least total P value of all pages assigned to the channel.

**HandleRemainingClients** Assigns remaining clients after all pages have been dealt with.

### Notes

The AllocateChannels function implements the PC algorithm. It first gets a list of all pages which are of interest to the clients. It then goes through the list, each time picking the page with the max PC value (*PickPageWithMaxPC*), choosing the least loaded channel (*GetLeastLoadedChannel*), assigning the page to this channel (*AssignPageToChannel*), and finally assigning the channel to all the clients interested in the page (*AssignChannelToClients*).

The algorithm terminates when either 1. All the clients have used up their channel quotas, or 2. All the pages in the database have been accounted for, or 3. Some pages are yet to be accounted for, but the clients who are interested in these pages have filled their channel quotas.

In the last two cases, if there remain any clients whose channel quotas are yet to be filled, then channels are randomly allocated to these clients (*HandleRemainingClients*).

The PC value of a page is the product of its P value (probability of generation at server) and its C value (number of clients interested in the page).

Note that pages being allocated to channels is just an internal step used by the channel assignment algorithm. It does not affect the page mapping decisions taken by the mapper.