Simulation of WAN Traffic

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

BY
P. N. SIREESH AND SMRUTI RANJAN SARANGI

UNDER THE GUIDANCE OF
PROF. S. P. PAL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITIUTE OF TECHNOLOGY
KHARAGPUR
MAY 2002

Certificate

This is to certify that the thesis entitled “Simulation of WAN Traffic”, submitted by P. N. Sireesh and Smruti Ranjan Sarangi for the award of the degree of *Bachelor of Technology* specialising in the field of *Computer Science and Engineering* from the Indian Institute of Technology, Kharagpur, is a record of an original work carried out by him under my supervision and guidance. The thesis has fulfilled all the requirements as per the regulations of the Institute and in my opinion has reached the standard needed for submission. The results embodied in this thesis have not been submitted to any other University or Insitute for the award of any degree or diploma.

Prof. S. P. Pal
Dated: May 2002

Acknowledgement

We wish to express our sincerest thanks and gratitude to our guide **Prof. S. P. Pal** for his expert guidance. His valuable suggestions were of immense help in completing the project work. They have been a constant source of encouragement and were always available in spite of his very busy schedule.

We are thankful to the department for providing such an amicable environment for research work. We thank all the faculty members, research scholars and non-teaching staff of the department for extending their full cooperation. We are thankful to our classmates who were very supportive.

We wish to express our deepest sense of gratitude towards our family and friends. Without their blessings and goodwill this project would never have been possible.

Abstract

This thesis presents the design of a WAN simulator named SWAN and related utilities. SWAN is an object-oriented simulator capable of simulating large scale networks with a heterogeneous community of users. It has a very flexible and customisable design. The simulator has a generic kernel that performs the functions of routing, multicasting, DNS and managing the components. There is a second layer that hosts the classes performing specific functions. These classes can be modified to suit the needs of various networks. Issues like portability and compatibility with other simulators has been taken into account. The input and output formats are standard languages like GML and XML. Along with the simulator this thesis describes a network diagram drawing tool called NetGraph that has a sophisticated drawing interface. The research contribution of the project lies in a Monte Carlo scheme called GRAPES for generating synthetic web requests. It combines two seemingly unrelated distributions and generates web requests that follow both of them. It is an order of magnitude faster than other request generators. Along with the sequential algorithm a work efficient parallel algorithm for the PRAM model was also developed.

Simulation of WAN Traffic

P.N.Sireesh and Smruti Ranjan Sarangi

Contents

List of Figures

Chapter 1

Introduction

The Internet has pervaded every aspect of human society. The World Wide Web (WWW) has become a major source of information, entertainment and a medium for business and commerce. The total number of web sites has grown from fifty in 1993 to fifty million by the year 2000. The phenomenal and explosive growth of the Web has sparked much research activity on improving the performance of the Web. Significant improvements have been realized in the performance of the Web due to the incorporation of newer network protocols [11] and web-caching algorithms [17, 10, 8, 4, 7], leading to better service to millions of users. For evaluation of such protocols and algorithms a realistic network environment must be created. Simulation is a natural scientific method in this direction. Several simulators like NS2 [13] and REAL [14] and artificial workload generators like ProWGen [6], SURGE [3], SpecWeb99 [15] and WebBench [16] have been used to create and simulate Web traffic at various levels in the network hierarchy. NS2 and REAL simulate traffic at the lower level whereas the others simulate traffic at the application level for wide area networks (WANs).

1.1 SWAN Simulator

This thesis describes the design of such sort of a simulator titled "SWAN" along with its associated utilities. SWAN has been designed to function as a WAN simulator. It simulates networks at a high level. As a consequence network layer and transport layer details are not handled. It just tries to simulate a network based on models of application level traffic adopting a macroscopic view of the WAN.

SWAN has chiefly three components.

- **Proxy** This represents a single user or a group of users generating web traffic. Such sort of traffic is mainly composed of http, ftp and smtp traffic. The proxy entity in SWAN represents the user in the inernet who continuously demands some service over the net. The behaviour of proxies has been studied in detail [2]. Several analytic models have been proposed to model such characteristics. The models chiefly concenterate on (i) The

popularity of the web pages requested, (ii) properties of temporal locality and (iii) rate of requests.

- **Server** This represents the web server. These entities host web pages. Proxies continuously request pages from these servers. Servers are distributed all over the internet and serve a variable number of pages with variable request rates. The characteristics of web servers have also been studied and several analytical models have been proposed on the basis of empirical observations. The key features studied in literature are (i) geographical distribution of servers (ii) distribution of file sizes (refer to [2]).

- **Cache** This entity lies between the web server and the proxy. It caches the web pages from the server. When the proxy requests for the page again then the request is satisfied from the cache. Thus the cache acts as an intermediary between the proxy and the server. There are several caching algorithms. Prominent among them are Harvest [8] and D4. There are also different cache replacement schemes like LFU, LRU, FIFO etc. This is probably the least studied among the three entities. The relationship of caching schemes to overall network performance is a big research problem.

SWAN is very different from other WAN simulators. Simulators like SURGE, ProWGen and SpecWeb either simulate the proxies or the servers, but not both. As an example SURGE generates proxy workloads but does not have the concept of servers that will serve the requests. SpecWeb simulates the web server and is geared towards measuring properties like server utilization and server load. SWAN subsumes all these functionalities and provides a holistic network environment possessing all the major components. As described in the previous paragraph it has users represented as proxies, servers and the cache. Thus, all network properties measurable in other network simulators can be measured in SWAN along with some extra properties of WAN's like latency and average cache hit ratios. Due to the holistic simulation environment of SWAN it is amenable to use in simulation of large wide area networks having a heterogenous community of users and web services.

During the course of the project the SWAN simulator was developed, coded, debugged and tested. The design of the simulator is presented in chapter 2. It is an object oriented design which is highly customisable depending upon the specific network being simulated. The architecture is basically broken up into two parts. One is the basic kernel that handles all the cumbersome jobs like routing, multicasting, DNS lookups etc. The implementers are not expected to change this code as it is common across most of the networks. But the flexibility of SWAN is owed to the second subsytems that consists of a group of abstract base classes. The implementers are expected to provide their own implementations. Thus it is a plug-in type of architecture where a new class with a well defined interface can be plugged in to change the behaviour of the simulator. Along with SWAN default implementations are provided. So the user just needs to change a few classes to tailor the simulator to the needs of

his network. To summarize, SWAN is a highly customisable, flexible object oriented simulator designed to simulate large scale networks.

During the design of SWAN certain issues like compatibility and portability were taken into account. SWAN must be compatible with other simulators. Along with that SWAN must be platform independent and must have hooks that can be used by visual programs to interact with the simulator. Thus, SWAN has functions that provide an interface to command line tools as well as other sophisticated visual tools. It was not always possible to make it completely platform independent due to the inherent limitations of the C++ langauge in which SWAN was coded. In such situations specific macros and functions were provided for different operating systems especially WIN32 and UNIX. As an example windows uses the MFC thread library whereas UNIX uses pthreads library. Thus two stub functions were provided such that programs could access SWAN in a transparent manner.

Now coming to the issue raised in the earlier paragraph related with compatibility of SWAN with other simulators. First of all the need was realized that probably an intelligent user would hack SWAN and other simulators and try to create a hybrid simulator optimized for his network. Keeping such sort of applications in mind SWAN has a very similar interface and component structure as other simulators. Since, the functionalities of most other simulators are a subset of SWAN's functionalities, SWAN can use them in place of its own classes. To use those simulators certain wrapper classes need to be written that conform to SWAN's specification. Another issue that crops up is how input/output will be handled. There is a standard format used by most network simulators called GML (Graph Markup Language). To maintain compatibility SWAN was made to communicate in GML. The network topology expressed in GML is fed to SWAN and it produces output in GML.

1.2 NetGraph

It was realised during the course of the project that complicated networks with a lot of properties need to be drawn. Thus a tool is required to facilitate drawing of complicated networks and ultimately saving it in a standard format like GML or XML. In response to this need a tool called NetGraph was developed in Visual C++. It features a rich graphical development environment. It has a lot of GUI functions as well as functions to manipulate network schemas. Thus very complicated networks can be drawn and manipulated efficiently with minimal effort. The final output of NetGraph is a GML file or XML page containing the network topology. NetGraph can also read GML and XML files. NetGraph has also been developed to have other uses. It is a general tool to draw a network. It has facilities of printing the network thus developed and it can save the drawing as a bitmap. NetGraph can even display the output GML file of other simulators very seamlessly.

1.3 GRAPES - a scheme for web request generation

As described above several utilities were required during the building of the simulator. Some of them were so significant that they became problems in their own right. As mentioned before analytical models of web traffic have been proposed on the basis of certain empirical observations. Now when web traffic is to be simulated a host of traffic generators are required that can simulate differents aspects of web traffic. Most of these generators generate random variables following a certain distribution. But one such generator that generates page requests turned out to be very complicated. It tries to combine distributions of page popularity and temporal locality. Several algorithms have been implemented in SURGE and ProWGen. But they were observed to have superlinear time complexity and were thus not scalable to large simulation environments.

So, the third problem was to design an efficient and scalable algorithm to generate web requests. This algorithm should obey the distributions of page popularity and temporal locality. A Monte Carlo scheme called GRAPES (Scalable Efficient Parallel Artificial Request Generator) was developed that possessed the above mentioned properties. Along with that it is amenable to a parallel implementation on the PRAM model that is work efficient. This scheme produces web request streams with properties very similar to those generated by other simulators. It is an order of magnitiude faster that other simulators for large inputs. This makes it highly scalable. If the resources of a single processor fall short then the parallel version can be used in a multiprocessor machine (refer to chapter 4.

1.4 Organization of the Thesis

In chapter 2 an overview of the SWAN simulator is presented. The object oriented design is presented in detail along with other interfacing details. Then certain experiments are described that were conducted on the WAN simulator for certain representative networks. The results are discussed along with proofs of correctness.

Chapter 3 discusses the NetGraph tool and gives a broad overview of its features. The interfacing details are discussed and some screenshots are presented. Chapter 4 describes the GRAPES scheme for request generation. There are a lot of intricate details that have been explained along with simulation results and their analysis. Chapter 5 concludes the dissertation and chapter 6 discusses the scope of future work.

Chapter 2

Simulator for a WAN - Swan

The World Wide Web is growing at an enormous rate. New strategies are required to support the growth and keep the latency of document retrieval within limits. Many simulators have been developed to simulate the internet at great detail (ns2 , real5). Our aim is to design a network simulator that simulates the internet at a high level and depicts its usage analytically. Several analytic models are used in building the Simulator. The simulator is used to reproduce Internet traffic with properties closely matching with those of the empirically observed properties. The Simulator can then be used for testing different network protocols, web-caching algorithms, planning an efficient Network topology, etc.

The Network topology of any WAN can be represented by a graph where a node represents each machine in the network and each communication link between two machines is represented by an edge. Each machine in the network can be a proxy server that generates requests on behalf of a community of users or a cache that stores some frequently referenced pages or a server that is the host of some pages. A proxy generates requests, and these requests are sent to the server. If on any intermediate node is a cache, and if the referenced page is present at the cache, then the cache sends the page to the proxy that has requested the page.

2.1 Architecture of the Simulator

This section describes the basic architecture of a WAN simulator. The design is an object oriented design which is completely customizable depending on the type of network being modeled. There are a set of core classes that provide the main functionality. Then there are a group of base classes which provide the advanced functionality. These base classes can be overridden to change the default behavior. This is a highly adaptive architecture that can support various types of networks, routing and caching strategies.

The top level view is described in figure 2.1.1.

The simulator has an array of nodes that emulate the network by passing messages. The messages are stored in an event queue. Depending upon the addressee of the message the appropriate node event handler is invoked. There

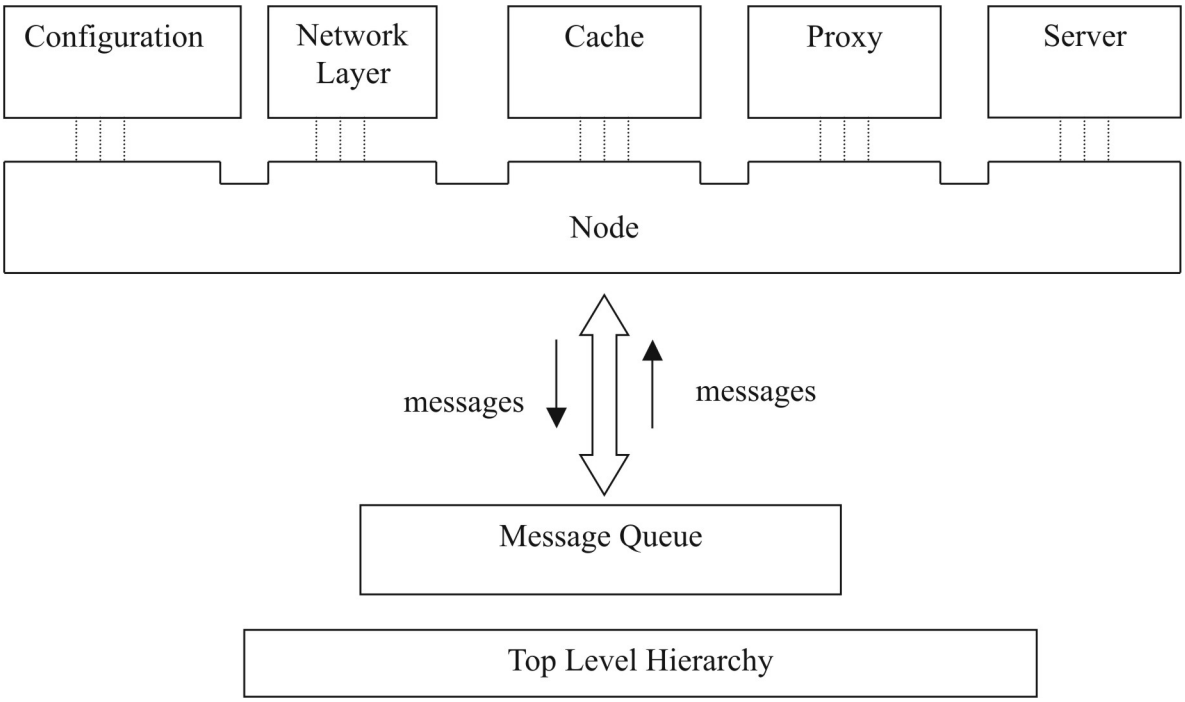

Figure 2.1.1: Top Level Hierarchy

are also two other modules that can be plugged into this architecture.
They are :

1. Simulation Visualiser
2. Module to write to stdout

The simulator can be represented as a set of functions built upon a message passing model (see figure 2.1.2).

2.1.1 Core Classes

There are four core subsystems that we wish to be part of every derived implementation.

1. Node
2. Message
3. Event
4. Dns

1. **Node**

```
class SimItem
{
```

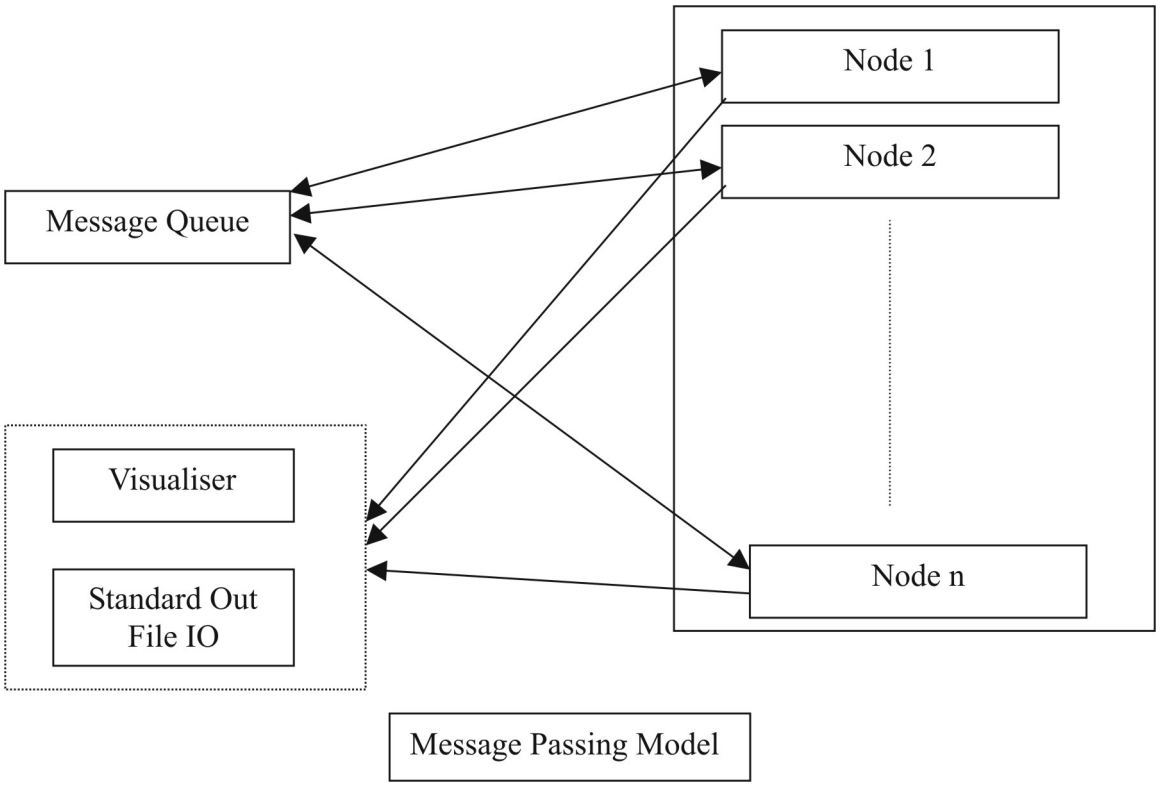

Figure 2.1.2: Message Passing Model

```
/// message processing function
virtual bool processMessage(Message* msg) = 0;
};

class Node : public SimItem
{
protected:
int id;

public :
    Configuration *config ;
    Cache *cache ;
    Network *network ;
    Proxy *proxy ;
    Server *server ;

    Node(int id) ;
    ~Node() ;

    bool processMessage(Message* msg) ;
    NodeStat* getStat() ;
    ConfigStat* getConfigStat() ;
```

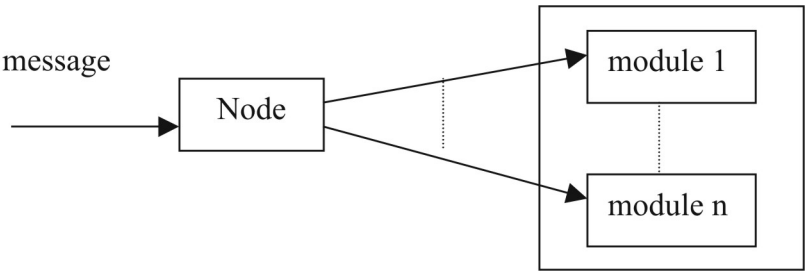

Figure 2.1.3: Message Passing

```
    CacheStat* getCacheStat() ;
    NetworkStat* getNetworkStat() ;
    ProxyStat* getProxyStat() ;
    ServerStat* getServerStat() ;
    int getId();
};
```

As this class is the only access point for external modules like the visualiser and the IO subsystem all its methods are public. As we don't intend to override the core, the functions are not virtual functions. All sub systems must implement the interface SimItem. It has just one abstract method. This method is the message processing function. The event queue will invoke this method for a node when it sees a message destined for it. The node will invoke this method for every subsystem. This allows more than one subsystem to process the same message. The return type is a boolean value which is true if the subsystem has completed processing the message and no more processing is required for the message. Figure 2.1.3 shows the message passing schema.

Each Node contains five sub systems - the Configuration, Network, Proxy, Server and Cache (see figure 2.1.4). For advanced functionality, any of these sub systems can be derived and plugged in to the Node. We will come to these sub systems later on.

2. **Message** This is a generic class that specifies the framework of a message. This class should be used for some simple cases as it provides a few default fields. It should be overridden when custom messages are required. This memo requires that the generic message type be typecast into the derived type in a subsystem that is willing to process it. The C++ <dynamic_cast> operator can be used for this purpose.

```
class Message
{
protected :
    int type ;
    int origSourceId ;
    int origDestId ;
```

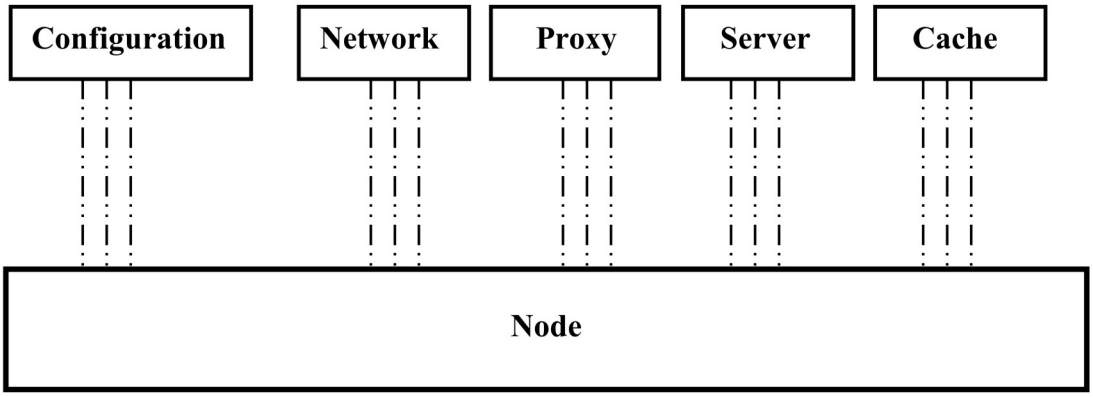

Figure 2.1.4: Node Class

```
    double origTime ;

public :
    void setType(int type);
    void setOrigSourceId(int id);
    void setOrigDestId(int id);
    void setOrigTime(double time);
    int field ;
    long size ;
    double nextTime ;
    int sourceId ;
    int destId ;
    Message(int type,int source,int dest,long size,int field) ;
    Message(int type,int source,int dest,long size,int field, double origTime);
    virtual ~Message() ;
    int getType() ;
    int getOrigSourceId() ;
    int getOrigDestId() ;
    double getOrigTime() ;
};
```

Every message has a type. This specifies the action to be done or re-

quested. The message types are defined in a separate header file "constants". In the current implementation there are six types of messages - PAGE_REQUEST, PAGE_REPLY, PROXY_WAKEUP, SOCKET _WAKEUP, DNS_REQUEST and DNS_REPLY.

In addition the message also contains information such as the original source of the message, the final destination of the message, the created time, the next occurrence time, the immediate source and the immediate destination.

The message also contains a field which contains the primary most important data. In case the message is required to carry more data, a void pointer can be used. The only condition is that the receiver should know what to expect and consequently typecast the data using the <dynamic_cast> operator.

3. **Event Queue** This is a global resource that is there in the global namespace Swan. There is a Event queue class that contains a priority queue of message pointers. The queue is prioritized based upon the next time that the message will be fired.

```
class EventQueue
{
public:
    vector<Message *> v;
    void push(Message *msg) ;
    Message* top() ;
    Message* pop() ;
    bool empty() ;
    void updateQueue(double time);
    EventQueue() ;
    virtual ~EventQueue() ;

private:
    int bsearch(int time, int start, int finish);
};
```

The Event Queue contains a vector which contains the messages. The messages in this vector are arranged according to their next occurrence time 'nextTime'. The message which is going to occur next i.e the message with the least next occurrence time will be at index 0 of the vector. When a message has to be pushed into the EventQueue, the message is inserted at the appropriate position in the vector so as not to change the order of the EventQueue. When a message is popped, the message at index 0 in the vector is removed.

4. **Dns** This class models a DNS server. The DNS server receives DNS requests for a certain page. The DNS server then sends a DNS reply to the node sending the DNS request the site at which the requested page can

be found. There are two associated functions. The first function returns the list of all the addresses and the second returns an arbitrary address among them. This class is really helpful to support mirroring of websites in servers etc. In the current implementation, we have taken that there is no time lag between the node sending the request and the node receiving the corresponding reply.

```
class Dns : public SimItem
{
protected:
    int numPages ;
    vector<int>* pages ;

public:
    void addEntry(int page,int server) ;
    void delEntry(int page, int server) ;
    int getSite(int page) ;
    vector<int>* getMultiSite(int page) ;
    Dns(int numPages) ;
    virtual ~Dns() ;
    virtual bool processMessage(Message* msg) ;
};
```

2.1.2 Description of the sub systems

Each Node class contains five sub-systems. Further sub systems can be added into the node class by deriving the node class.

Each node class consists of the five following sub systems.

1. Configuration
2. Network
3. Proxy
4. Server
5. Cache

1. **Configuration** Depending upon the type of the network a node might have a lot of configuration information. This information might contain the network addresses of the node, hardware addresses of the node etc. Along with this we also need to know whether the node is configured as proxy, server or cache or as a mixture of any of the three. To keep all this information the Configuration class is used. This has the proxy, server, cache configuration information along with the unicast id and the multicast id's of the node.

```
class Configuration
{

protected :
     bool proxy ;
     bool server ;
     bool cache ;
     int id ;
     char desc[64] ;
     vector<int> multids ;

public:
     Configuration(int id,char* desc,bool proxy,bool server,bool cache) ;
     virtual ~Configuration() ;
     bool getProxy() ;
     bool getCache() ;
     bool getServer() ;
     int getId() ;
     char* getDesc() ;
     vector<int>* getMultids() ;
     void addId(int id) ;
     void delId(int id) ;
     bool exists(int id) ;
     ConfigStat* getStat() ;
};
```

2. **Network** This is by far the most important class in the simulator. It simulates the network layer functions :

   - Routing
   - Link Management and revival
   - Unicasting and multicasting
   - Flow Control

   This is an ideal situation to use the marvels of object oriented programming. The Network contains the router, the socket and the adjacent links. The router and socket implement the SimItem interface. When the network receives messages to be processed, it passes on these messages to the router and the socket for processing. The network class has the same plug in architecture as described earlier. Its high level diagram is shown in figure 2.1.5.

```
class Network : public SimItem
{

protected:
```

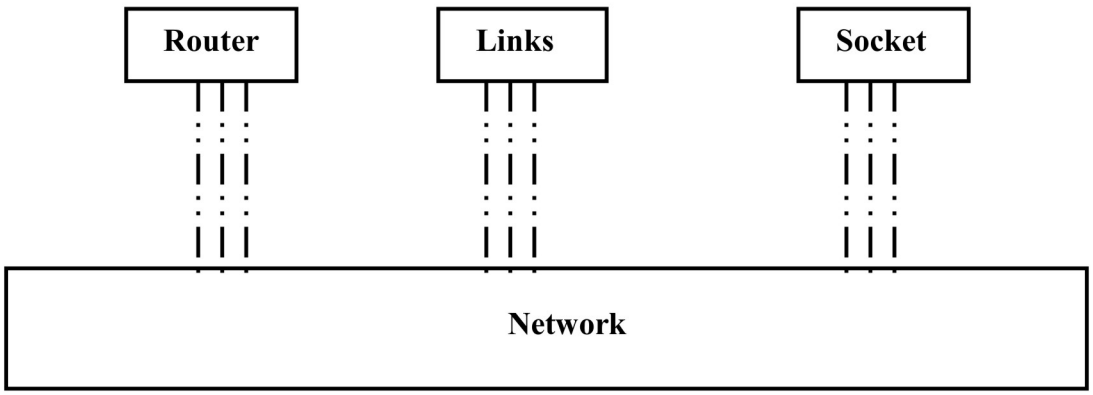

Figure 2.1.5: Network Class

```
   int id;

public:
   Router* router ;
   vector<Link*> links ;
   Socket* socket ;

   int getId();
   virtual NetworkStat* getStat() ;
   SocketStat* getSocketStat();
   RouterStat* getRouterStat();
   virtual bool processMessage(Message* msg) ;
   Network(int id);
   virtual ~Network() ;
};
```

Default implementations are provided for these subsystems. For the router static shortest path min-hop routing is used.

An array of link ids is stored. These pointers to links have the bandwidth and latency information. Useful extension to these classes can have error information and other features.

The Socket class is a class that is used to transmit messages (unicast

or multicast) to other nodes of the network. This is very similar to the send system call that sends a group of bytes over the network. This incorporates data link, network and transport layer functionality. In the default implementation this gets the routing pathfrom the router and unicasts or multicasts the packets to their desired destination. But more advanced implementations like introducing errors, error control, sliding window etc. are possible. This socket can even act like a Tcp or Udp socket that has sliding windows and the slow start mechanism. In the base class, the socket receives the message from the network after the message has been processes by the router. Therefore, the destination field of the message is already set. The socket then immediately starts transmitting the message across the link to the destination if the link is free. In case, the link is busy, the socket stores the message in its internal message queue, and when the link becomes free, the socket starts transmitting the message.

(a) **Router** The router is a base class that provides shortest path min-hop routing facilities.

```
class Router : public SimItem
{
protected:
    int id;
    int *table ;
    int tableSize ;

    virtual int getNextNode(int dest) ;
    virtual int getNextLink(int dest) ;
    virtual vector<int>* getNodes(vector<int>* dest) ;
    virtual vector<int>* getLinks(vector<int>* dest) ;

public :
    virtual bool processMessage(Message* msg) ;
    Router(int id, int tablesize) ;
    Router(int id) ;
    virtual ~Router() ;
    RouterStat* getStat();
};
```

The router also derives the simItem interface. The routing subsystem can process some messages. The messages that signal link failure and link revival should be processed by the router. Then the router can update its routing tables based on the new information obtained. In the base class, when the router receives any message which has to be routed, it sets the destination field of the message and passes it back to the network.

The router has support for unicasting. Given the destination, the functions getNextNode and getNextLink return the next node and

link on the way. If the destination is the same as this node then the next node is the id of the node and the next link is -1. If a path does not exist then the next link and node are both -1. The router supports multicasting. Given a vector of nodesm the methods getNodes() and getLinks() returns a vector of next nodes or links to take to reach the destination.

(b) **Link** Next the link class is described. Each link has a unique linkId. Along with an unique id the link class stores information about its bandwidth, delay and its present status(activated or cutoff). The units of bandwidth and delay are in bytes per second and seconds respectively but any other unit can be chosen.

```
class Link
{
protected:
    int id ;
    int nodeA ;
    int nodeB ;
    double timeBusyUntil;
    void setTimeBusyUntil(double time);

public :
    double bandwidth ;
    double delay ;
    bool status ;
    long bytesAB;
    long bytesBA;
    int getId() ;
    int getNodeA() ;
    int getNodeB() ;
    double getTimeBusyUntil();
    void transmit(int src, double time, long size);
    Link(int id,int nodeA,int nodeB,double bandwidth,double delay,
                                            bool status= true) ;
    virtual ~Link() ;
    virtual LinkStat* getStat() ;
};
```

(c) **Socket** This is the subsystem that transmits data to an adjacent node. This class is meant to emulate a generic socket class that can transfer a group of bytes from one end to the other. In the default implementation unicasting and multicasting are supported. Issues like error control, congestion handling etc. are not handled by the default implementation. For custom sockets, this class cen be extended to provide advanced functionality. For eg. this class can be extended to contain a sliding window and congestion handling routines.

In the default implementation this class looks like this:

```
class Socket : public SimItem
{
protected :
    int id;
    MessageQueue* queue ;
    int maxlen ;

public :
    long reqSent;
    long packDropped;
    virtual void unicast(Message* msg) ;
    virtual void multicast(Message *msg) ;
    int getAdjacentLink(int dest);
    virtual bool processMessage(Message* msg) ;
    Socket(int id, int maxlen = 100) ;
    virtual ~Socket() ;
    SocketStat* getStat();
};
```

The socket also contains an internal message queue. When messages need to be buffered, the messages are stored in this message queue. In the base class, the message queue is implemented as a FIFO message queue.

The function unicast(Message* msg) transfers the message from the current node to the adjacent node. The destId field of the message gives the id of the next node.

3. **Cache** A node might be configured as a cache. The job of this cache object will be to temporarily cache web objects. Older objects will be replaced with new ones. Upon a cache miss the cache might have several strategies of fetching the page. The cache might query the parent or it might multicast the request to its group members. After getting the page the cache subsystem decides whether to cache the page or not. There might be several schemes for this LRU, LFU etc. Thus a base cache object has been created which is extensible to embody one or more of these schemes. The default implementation implements the LRU cache with no caching algorithm. The class design for the cache object is presented below :

```
class CacheType
{
public :
    int rep;
    int algo ;
    CacheType() ;
```

```
    CacheType(int rep,int algo) ;
    virtual ~CacheType() ;
};

class Cache : public SimItem
{
protected:
    int id;
    Network *net;
    vector<int*> pages ;
    long totHits;
    long totMisses;
    int getPageSize(int page);
    int totPages ;
    long totBytes ;
    int getNumBytes();
    CacheType *type ;
    virtual bool checkPage(int page) ;
    virtual bool accessPage(int page) ;
    virtual bool fetchPage(int page) ;
    virtual bool addPage(int page, int size) ;
    virtual bool flushPage(int page) ;
    virtual bool flush(int numpages = 1) ;

public :
    virtual bool processMessage(Message* msg) ;
    virtual CacheStat* getStat() ;
    Cache(int id, CacheType* type, int totPages = -1, long totBytes = -1);
    virtual ~Cache() ;
};
```

The most important attribute of the cache is its size. The size is specified in the number of pages or in the number of bytes or both. The size in number of pages is sometimes important for simulation purposes. The default values for these are -1. The value of -1 signifies infinite size. Setting -1 as the maximum number of pages means that there is no limit on the number of pages as long as the total byte count constraints are satisfied. Setting -1 as the maximum no of bytes in the cache means that there is no limit on the size of the cache as long as the total no of pages constraint is not violated.

A data structure called CacheType is defined. Caches have two main features. One is the cache replacement algorithm and the other is the type of algorithm used to fetch pages and disseminate cache information. So CacheType has two integer fields rep and algo. rep specifies the specific replacement algorithm and algo the caching strategy. Generic functions are provided to access, check and delete pages in the cache. These methods

can be overridden to code different algorithms. The function checkPage() checks whether a page is in the cache or not. accessPage() accesses the page if the page is present in the cache. In the base class, accessPage simply makes the given page the most recently used page in the cache. fetchPage() fetches a page from a distant server if the page is not present in the cache. In the base class, the fetchPage is empty and does not provide any functionality. Hence, in the base class, if the page is not present, the cache will not completely process the message, and the message will be handed over to the network which will transmit the message to its proper destination. addPage() will add the page into the cache. flushPage will flush a particular page or certain no of pages from the cache. In the base class, flushPage will flush the least recently used pages from the cache.

In the base class, LRU strategy is used to replace the pages, and Harvest diffusion is used. This class has been extended to make a class called LFUCache. The LFUCache uses LFU for page replacement and Harvest diffusion.

```
class LFUCache : public Cache
{
protected:
    vector<long *> count;
    virtual bool checkPage(int page) ;
    virtual bool accessPage(int page) ;
    virtual bool fetchPage(int page) ;
    virtual bool addPage(int page, int size) ;
    virtual bool flushPage(int page) ;
    virtual bool flush(int numpages = 1) ;
    void flushContents();
    void incrCount(int page);
    long getCount(int page);
    bool isPageCacheable(int page, int size);

public:
    LFUCache(int id, CacheType* type, int totPages = -1, long totBytes = -1) :
    Cache(parent, *net, type, totPages, totBytes);
    virtual bool processMessage(Message* msg) ;
    virtual CacheStat* getStat() ;
    virtual ~LFUCache();
}
```

4. **Proxy** This class models the proxy server in the network. This class sends requests for pages. This request stream should be in accordance with empirically observed distributions.

```
class Proxy : public SimItem
{
```

```
protected :
    int id;
    double interval ;
    double mean ;
    double var ;
    double max ;
    long reqSent;
    long reqRecvd;
    long totBytes;
    double totLatency;
    vector <int *> requests;
    virtual int sendNext() ;

public :
    ReqGen *reqGen ;
    TraffGen *traffGen ;
    virtual bool processMessage(Message* msg) ;
    virtual ProxyStat* getStat() ;
    Proxy(int id, double interval, double mean, double var, double max);
    virtual ~Proxy() ;
    int getId();
};
```

After the Network, the Proxy class is the next most important class. This class generates the requests and sends them to the network for transmission. The proxy sends a specific number of requests per interval. The mean, variance and maximum number of requests generated in an interval and the length of the interval are specified by the user. After getting the number of requests from the traffic generator this class gets the requests from the request generator. Then these requests are transmitted to the network. To summarize the Proxy server generates the number of requests to be sent in an interval, gets the specified number of requests from the request generator and transmits them over the network. If it is not successful in doing so then a congestion control routine should be invoked. Such a routine is not a part of the default implementation. This routine should also be invoked upon receiving a congestion message from the point of congestion ie. a distant router. The two major sub systems of the proxy class are the :

(a) Traffic generator

(b) Request generator

The architecture of the proxy server is shown in figure 2.1.6.

(a) **Traffic Generator** This class is responsible for generating the number of requests sent per interval by the proxy server. In the base class, the traffic generator incorporates the code of generating the

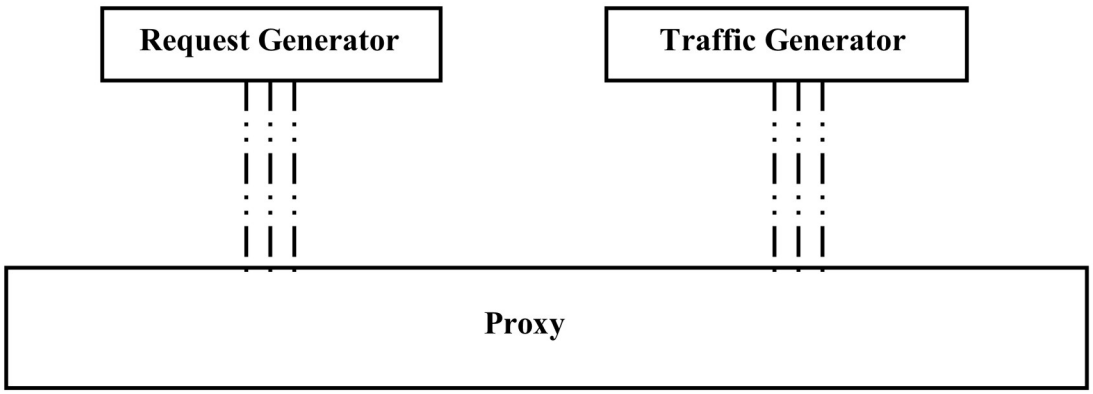

Figure 2.1.6: Proxy Class

fractional guassian noise which was downloaded from the Internet. But for different situations different traffic generators can be used. The traffic generator cen be extended to produce different varieties of traffic.

```
class TraffGen
{
private:
    vector<int> *buffer;

public :
    double mean ;
    double var ;
    double max ;
    virtual int getNext();
    virtual void flush() ;
    TraffGen(double mean,double var, double max) ;
    virtual ~TraffGen() ;
};
```

The traffic generator uses an internal buffer. Usually traffic generators do not produce the no of requests on the fly, but generate a block of values. These block of values are stored in the buffer. When the

buffer empties, the traffic generator generates a new block of values. The traffic Generator contains two virtual methods - int getNext() - this returns the no of requests to be generated in an interval and void flush() - this empties the buffer that is used.

(b) **Request Generator** This class yields the request for the next page. The request generator must incorporate the Zipf distribution of page popularity and some model of temporal locality. In the base class, only Zipf distribution is obeyed and no form of temporal locality is implemented.

```
class ReqGen
{
private:
    vector<int>* buffer;
    vector<double>* prob;
    int sigma;

public :
    int numPages ;
    ReqGen(int numPages, int sigma) ;
    virtual ~ReqGen() ;
    virtual int getNext() ;
    virtual void flush() ;
};
```

The Request Generator takes as a parameter the number of pages. The pages are numbered from 1 to numPages. The function getNext () returns the number of the next page to be requested. This number lies between 1 and numPages.

5. **Server** This is a small class that stores the list of pages available at the node. It answers queries about page existence and has methods to fetch a page. It has one subsystem - the Size generator. This module generates the sizes of the pages stored in the server. For this purpose a SizeGen class is proposed which takes as input a page id and returns its size in bytes. The architecture is shown in figure 2.1.7.

```
class Server : public SimItem
{
protected :
   vector<int> pages ;
   int id;
   int requested;
   bool fetchPage(int p) ;

public :
   SizeGen* sizeGen ;
```

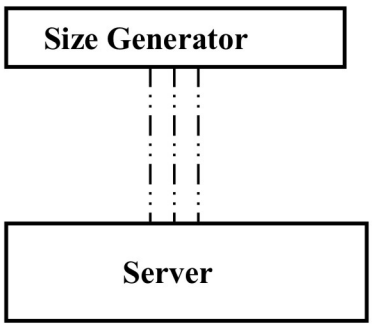

Figure 2.1.7: Server Class

```
   Server(int id) ;
   virtual ~Server() ;
   virtual bool processMessage(Message* msg) ;
   virtual ServerStat* getStat() ;
   int getNumPages() ;
   vector<int> getPages();
   void addPage(int p) ;
   bool pageExists(int p) ;
};
```

The pages are stored in a vector. The number of pages is stored in the variable numPages. The function fetchPage fetches a page from the server and returns a status code. If it was successful it returns true else returns false.

(a) **Size Generator**

```
class SizeGen
{
protected:
    int numPages ;

public:
    virtual int getSize(int page) ;
    SizeGen(int numPages) ;
```

```
    virtual ~SizeGen() ;
};
```

This is a base class for the implementation of a size generator class. This takes the page id and returns the size of the page. In the base class, the size generator returns a constant page size which is defined in "constants.h" as PAGE_SIZE.

2.1.3 Statistics

The simulation can be seen by a visualiser as it progresses or periodically the output can be written to standard out. The point to access this functionality is the Node object's getStat method. This function returns a pointer to an object of type NodeStat which is just an aggregation of Stat objects of the component subsystems. This class is as follows :

```
class Stat
{
public :
    Stat() ;
    virtual ~Stat() ;
    virtual void print();
    virtual void print(ofstream* out);
};
```

The class Stat is the base class for all stat objects. It has been kept empty in this implementation but functions may be added later on. All objects that denote some form of statistics are required to extend this class. The component objects of the NodeStat class are described next.

```
class NodeStat : public Stat
{
public:
    int id;
    char desc[128];
    ConfigStat* configStat;
    NetworkStat* netStat ;
    CacheStat* cacheStat ;
    ProxyStat* proxyStat ;
    ServerStat* serverStat ;
    NodeStat() ;
    virtual ~NodeStat() ;
    virtual void print();
    virtual void print(ofstream* out);
};

class ConfigStat : public Stat
{
```

```
public:
    int id ;
    char* desc;
    vector<int> multids ;
    bool proxy ;
    bool server ;
    bool cache;
    ConfigStat() ;
    virtual ~ConfigStat() ;
    virtual void print();
    virtual void print(ofstream* out);
};

class NetworkStat : public Stat
{
public :
    SocketStat* socketStat;
    RouterStat* routerStat;
    int numLinks ;
    NetworkStat() ;
    virtual ~NetworkStat() ;
    virtual void print();
    virtual void print(ofstream* out);
};

class ProxyStat : public Stat
{
public :
    long reqSent ;
    long reqRecvd ;
    double avgLatency ;
    long totBytes;
    double mean ;
    double interval ;
    double var ;
    ProxyStat() ;
    virtual ~ProxyStat() ;
    virtbual void print();
    virtual void print(ofstream* out);
};

class CacheStat : public Stat
{
public :
    long totHits ;
    long totMisses ;
    CacheStat() ;
```

```
    virtual ~CacheStat() ;
    virtual void print();
    virtual void print(ofstream *out);
};

class ServerStat : public Stat
{
public :
   int pages ;
   long requested ;
   ServerStat() ;
   virtual ~ServerStat() ;
   virtual void print();
   virtual void print(ofstream* out);
};
```

These Stat objects keep the full state of the simulation. Whenever the visualiser wishes to know the state of the simulation it can get the NodeStat object and read the simulation variables of its components objects. If the implementer wishes to send the output to stdout then he can write a function that sends the contents of the NodeStat object to stdout. The Stat objects discussed above are for the Node class and its components. But we also need to keep some statistics for the link class as well. This information will contain the number of bytes sent on a link in each direction. The link class has an associated stat object called LinkStat.

```
class LinkStat : public Stat
{
public :
    int id ;
    int nodeA ;
    int nodeB ;
    double bandwidth ;
    double delay ;
    long bytesAB ;
    long bytesBA ;
    LinkStat() ;
    virtual ~LinkStat() ;
    virtual void print();
    virtual void print(ofstream* out);
};
```

2.1.4 Global Data Structures

Information about the network topology, sizes and location of pages etc. is required by various classes all across the class hierarchy. One design option is to have these as the static members of a class. This will encapsulate the names

of the data structures but does not provide any other functionality. Another option can be to use them as global data structures. Then we don't have any encapsulation. To solve this problem it was decided to keep all these data structures and their associated functions in a namespace called Swan. This provides encapsulation as well as global access.

```
namespace Swan
{
   double haltTime;
   vector<Node *> simNodes ;
   vector<Link *> simLinks ;
   int **simGraph ;
   int simPages ;
   Dns* simDns ;
   MultiCastServer* multiCastServer ;
   EventQueue* simMsgQueue ;
   double simTime ;
   simstat simStatus ;
};
```

The Swan namespace contains the list of nodes and links in the network. It also contains other global information such as no of pages, the current simulation time, the halt time of the simulation. The DNS server and the multicast server are also contained in the Swan namespace.

2.2 GML File Format

The data required for a simulation will be read from a file. This Input file must contain information about the network topology, the node parameters (cache size, request rate etc.) and the link parameters (bandwidth, delay etc). The data required must be specified in some format. After the simulation is over, the results will be written on to another file. This output file contains information about the network topology and the node statistics (total hits, total misses, latency etc) and the link statistics (total bytes transmitted in either direction etc).

In our simulator, the input file and the output file are specified using the Graph Modeling language (GML). GML's key features are portability, simple syntax, extensibility and flexibility. A GML file consists of hierarchically organized key-value pairs. A key is a sequence of alphanumeric characters, such as graph or id. A value is either an integer, a floating point number, a string or a list of key-value pairs enclosed in square brackets. GML can represent arbitrary data, and it is possible to attach additional information to every object. Graphs are represented by the keys graph, node and edge. The topological structure is modeled with the node's id and the edge's source and target attributes: the id attributes assign numbers to nodes, which are referenced by source and target.

A graph is defined by the keys graph, node and edge, where node and edge are sons of graph in no particular order. Each non isolated node must have a

unique .graph.node.id attribute. Furthermore, the end nodes of the edges are given by the .graph.edge.source and .graph.edge.target attributes. Their values are the the .graph.node.id values of end nodes. There are only two restrictions for graphs:

1. The values of the .graph.node.id elements must be unique within the graph.
2. Each edge must have .graph.edge.source and .graph.edge.target attributes.

A template shown below gives the structure of this file.

```
graph [
    directed 0
    node [
        id 1
        <key_name>  <key_value>
        <key_name>  <key_value>
        ...
        ...
        ...
        <key_name>  <key_value>
    ]
    node[
        ...
        ...
    ]

    link [
        id 10
        source 1
        target 2
        <key_name>  <key_value>
        <key_name>  <key_value>
        ...
        ...
        ...
        <key_name>  <key_value>
    ]
]
```

There shall be as many node keys as the number of nodes in the network and as many link keys as the number of links in the network. Attribute values will include information such as node (or link) identifier , caching policy, proxy request rate etc in the case of the input GML file and total hits/misses, latency etc in the case of the output GML file.

2.3 Simulations and Results

We have conducted several Simulations on various network topologies.

The requests generated by a Proxy for a page follows the Zipf's law. The Zipf's law of relative page popularity states that the probability that the request is for a given page is inversely proportional to its relative page popularity. The requests do not follow any model for temporal locality.

The number of requests generated by a proxy in an interval follows the self-similar traffic distribution.

The caching algorithm used is Perfect - LFU. In Perfect LFU, the cache maintains an access count for every page. Every time the page is referenced its access count is incremented. Whenever a page needs to be replaced in the cache, the page with the least access count is replaced.

In a network topology consisting of caching servers which implement the Perfect LFU, the caching servers in the network attain a stable state, i.e. when the cache contents of all the caches in the network become fixed. The cache receives requests for several pages. In the steady state, the pages which were referenced the most at the cache will be stored.

All the pages are assumed to have a fixed page size. Routing is done using shortest path - min hop routing.

Next we will discuss the results of our simulations on various topologies.

2.3.1 Line Topology

In this case, we assume that there are n cache servers connected in a line with a proxy at one end and a server on another end (refer to figure 2.3.8). The server serves as the central repository of all the pages. Since it is assumed that the size of the pages is fixed, we can represent the size of the cache as the number of pages it can store.

We denote the cache size at the l'th level as C_l. The proxy is at level 1 and as we move towards the server the level increases by one. In steady state when the cache contents tend to be static, the lowest level cache will contain the C_1 most popular pages. In the second level cache, the next C_2 most popular pages will be stored. This is because when the level 1 cache gets filled up with the C_l most popular pages, then the requests for those pages will be satisfied in the lowest level cache only. So the requests that will traverse to level 2 cache will be for pages whose popularity index is greater than C_l. Extending the steady state assumptions to all levels of cache, the cache at level l will stores those pages whose popularity index is between $(C_l + C_2 + \ldots C_l - 1)$ and $(C_l + C_2 + \ldots C_l - 1 + C_l)$.

Simulation Results - In our simulation, we have taken a line topology consisting of a server at one end, and 3 caches connected in a chain. There is a proxy at the other end. We ran simulation for 100 pages in the origin server and set the maximum average request rate to .01 requests per second. The size of the pages was taken as 10KB. The simulation was done for a total time of 100 hours. All the caches were taken to store a maximum of 10 pages.

At the end of the simulation, the cache contents and the access counts of

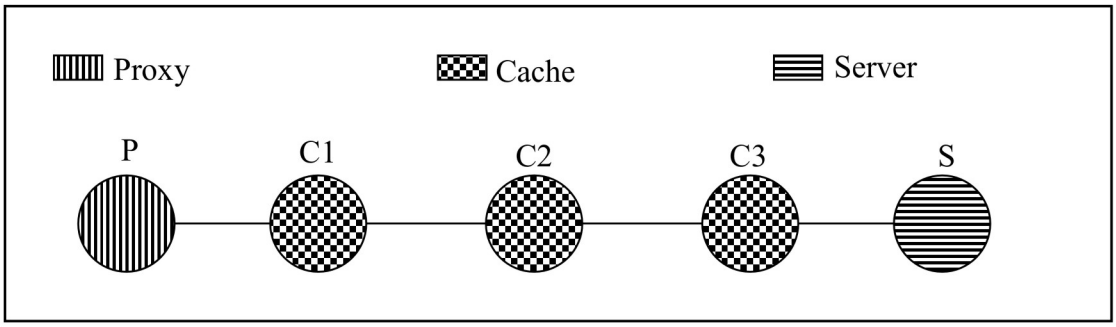

Figure 2.3.8: Line Topology

all the pages at each cache was recorded. The observed results matches closely with the expected results.

The cache C1 cached the 10 most popular pages, cache C2 cached the next 10 most popular pages and cache C3 cached the next 10 most popular pages.

2.3.2 Complete Binary Tree Topology

In this topology, the server is at the root of a complete binary tree of caches, terminating at the leaf nodes (refer to figure 2.3.9). Each leaf node is a proxy. We make a set of simplifying assumptions. We assume that all the proxies are equally active, and have the same request rate for all the pages. Based on this assumption, we set equal cache sizes for all caches at the same level in the binary tree. Let us denote the size of the cache at the l'th level in the binary tree to be C_l.

When all the caches attain a steady state, the lowest level cache will be filled with C_1 most popular pages. The next higher level cache will receive requests for pages whose popularity index is greater than C_l. So, at steady state, the next higher level cache will get filled with the next C_2 most popular pages. The contents of the caches in the case of this tree topology, is similar to that of the line topology. Since the cache sizes at any particular level are identical and all the proxies have the same behavior, all caches at the same level in the tree would have similar steady state conditions.

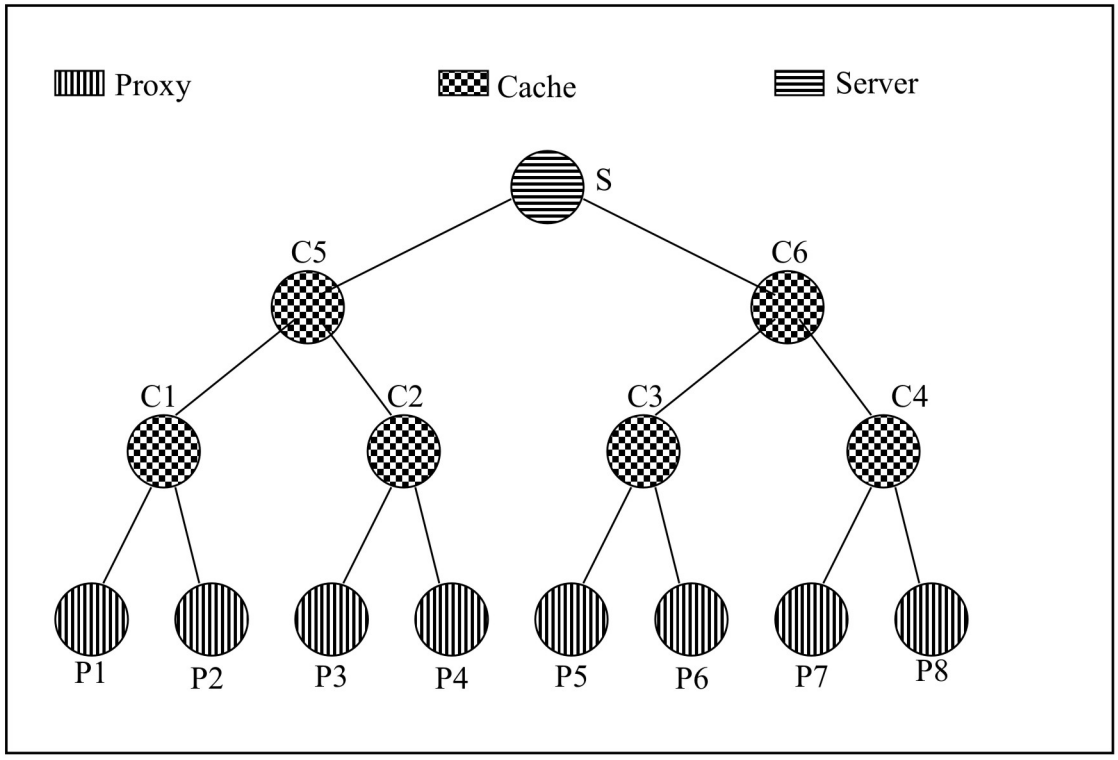

Figure 2.3.9: Complete Binary Tree Topology

Simulation Results - In our simulation, we have taken a complete binary tree with height 3. The root of the binary tree is a server which contains all the pages. The leaf nodes are the proxies, and the remaining nodes are caches. We ran simulations for 100 pages in the origin server and set the maximum average request rate to .01 requests per second. The size of the pages was taken as 10KB. The simulation was done for a total time of 100 hours. All the caches were taken to store a maximum of 10 pages.

At the end of the simulation, the cache contents and the access counts of all the pages at each cache was recorded. The observed results matches closely with the expected results.

The lowest level caches C1,C2,C3 and C4 cached the 10 most popular pages. The next 10 most popular pages were cached in the higher level caches C5 and C6.

Chapter 3

NetGraph

During the course of making the SWAN simulator a need was felt for a tool that facilitates drawing network topologies. There are several commercial products that provide a visual interface for drawing networks and even save it in GML. But all the products surveyed suffered from certain deficiencies. Mainly they are geared for certain kinds of networks. They are not general enough to support any kind of network. Along with that they don't have functionalities to manipulate properties of network elements. As an example a server might have certain properties like : number of web pages, ip address etc. For some tools these properties were fixed and no new property could be added and for some others this information could not be stored or used across multiple files. This led to the development of a network drawing tool that aimed at removing all this deficiencies and to be as generic as possible.

3.1 Overview of NetGraph

NetGraph is a network drawing tool coded in Visual C++. The next subsection presents an overview of its basic building blocks.

3.1.1 Basic Buiding Blocks

There are two basic entities called a node and a link.

Node A **node** represents any network entity. It can be a proxy server generating requests, a web server, a caching server, DNS server, Multicast server, switch, router or any other active or passive entity in a network. Depending upon what the node stands for it can have different properties. As an example if the node is a web cache then it will have the following properties: cache size, cache replacement policy, resource reservation scheme etc. Thus every type of node has a list of properties associated with it. Now a network has several types of nodes. Each node has its associated properties.

Link A **link** is a communication medium between two nodes. As an example a ethernet line between two machines is a link. A fiber optical connection between two routers is a link etc. Links need not be wires or other

hardware entities. It is possible to have wireless links eg. satellite communication. As nodes can be different links can also be different. A FDDI link has different properties as compared to an Ethernet link. Thus, each link also has a list of properties associated with it. For a standard link like a telephone line some properties are bandwidth, latency and error probability. For mobile links congestion and interference are very important factors.

A network is a combination of nodes and links. Nodes and links can be arranged in various configurations. This defines the network topology. The nature of the network is defined by the properties of the nodes and links. In this context two definitions are presented that differentiate between these two facets of a network.

Schema The list of the type of nodes and links in a network along with their properties is called a **schema**. The schema defines the type of a network. For example the internet has a certain schema, LAN's have a certain schema and home networks on DSL have another schema. For similar networks the schema is the same. NetGraph explicity realizes this and there are facilities to export and import a schema.

Drawing Given a schema nodes and links can be arranged in all possible permutations. Each such permutation corresponds to a **drawing** of a network. A drawing defines the network topology. The nodes correspond to vertices of a graph and the links correspond to directed edges.

Thus, an instance of a schema and a drawing completely specify a network upto isomorphism. NetGraph takes care of these two entities separately. At the outset the user creates a schema or imports one if he already has one. He also has the option of editing the imported schema. Once the schema is built the types of nodes and links are known and he can start drawing the network. While drawing the network the user can edit the schema and export it such that another user can use the schema for drawing his network. Each kind of a network is expected to have its own schema. The format of the schema has been designed to be compatible with GML. GML format recognizes four kind of objects. They are int, real, string and list. The list is a linked list of key value pairs of GML objects. Lists are not supported in this version of NetGraph. So NetGraph supports three kinds of properties. They can be integers, real numbers or strings. For each object the schema maintains a list of properties. These are key value pairs. The key is the name of the property and the value is its type (int, real or string). The collection of all these key-value pairs defines the schema. For use in a visual environment there is another property which specifies the colour of the node or the link when it would be drawn.

After the schema comes the drawing. The drawing interface of NetGraph is very similar to that of popular windows softwares like MS-Word or Corel Draw. Objects can be drawn, resized and a group of objects formatted. There are also facilites for the standard operations of cut,clear,copy and paste across applications. The drawing is transformed to its GML representation and is

then sent to the clipboard. Thus if a NetGraph drawing is pasted in another instance of a NetGraph application then the result is a drawing otherwise it is plain text in the form of a GML file.

While the user is drawing he needs to populate the instance of the schema associated with the drawing. That means that he must fill in the properties of every node. As an example if there is a node of type server with two properties ie. ip_address and location. After the user draws a server node he needs to fill in the ip_address and location of the node. A drawing is complete when all its constituent nodes and links have been drawn and their property fields have been filled up. Thus, a specific network consists of two parts. The drawing and the instance of the schema. The NetGraph file format (.ntg) which is a windows binary contains three pieces of information. They are as follows :

1. Schema of the network - This can also be exported separately as a schema file (.sch).
2. Instance of the schema - These are the values of the properties of each node and link.
3. Drawing - This consists of the geometric positions of the nodes and links. It contains some extra information about the links. It stores the id's of the two endpoints of a link and whether it is directed or undirected.

3.1.2 Basic Functionalities

The two major subsystems of NetGraph are :

- Schema Editor
- Drawing Editor

The schema editor supports entering of schemas. Along with these details the colour of the nodes and links are also entered. This is used when the user starts drawing the network. The schema editor uses the ActiveX control MS-FlexGrid. The functionalities of this control are harnessed in taking a schema as input.

After the schema is entered the user is all set to draw the network. The user can choose to a link, a node or the empty pointer. By choosing a node or a link he can draw a network by standard drag and drop techniques. If he chooses the empty pointer then he can select objects or a group of objects. Then he can move, resize or reorient them. In this mode he can change the properties of a network entity (node or link) by double clicking on it. The two subsystems are described in the following sections along with details of implementation and screenshots.

3.2 Schema Editor

The application starts out by displaying the schema editor. This is shown in figure 3.2.1.

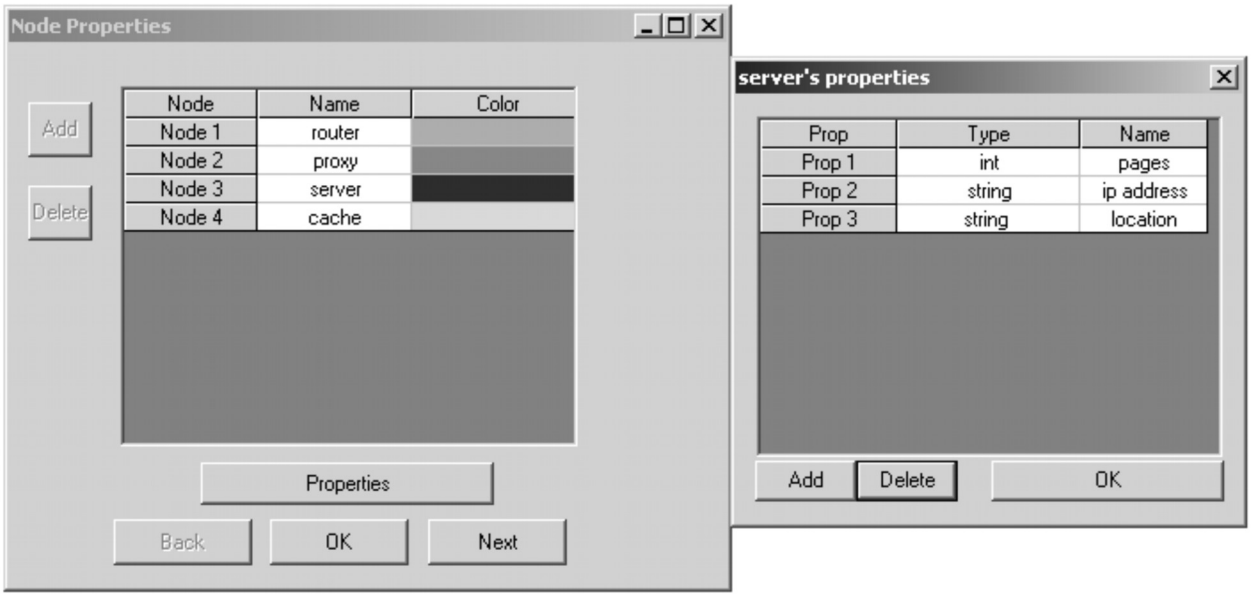

Figure 3.2.1: Schema Editor

3.2.1 Features

The first window in figure 3.2.1 shows the node properties window. In this window the user specifies the list of nodes. He specifies the name of the nodes and beside the name of the node he specifies the colour of the node. He can edit the list by adding or deleting nodes. Whenever a node is added or deleted the serial numbers automatically readjust themselves. There is a similar interface for entering the names and colours of links. The user can swith between the node and link windows by the next and back buttons. This defines one hierarchy of the schema. For the properties associated with each node or link there is a properties button. Upon clicking it a new window pops up where the user specifies the list of properties and their respective types. It has the same interface as the former window supporting editing of the property list. In the specific example the properties of the server node are being entered. In a similar way the properties of all nodes and links can be entered in the schema editor. This schema can be saved for use in later applications. There is a facility to import the schema in the File menu. Then this process of entering the schema is circumvented. Once the schema is made the user can proceed to make the drawing. But there are several important things that he can do before that. He can save the schema as a (.sch) file which is a windows binary or he can export it in gml or xml format.

3.2.2 Implementation

The grid that is visible is an ActiveX control called MSFlexGrid. The data is entered through standard methods of the ActiveX control. The data is mirrored in data structures. The important data structures are as follows.

tuple

This consists of a key-value pair. Both of them are strings. It has these two fields along with their getter and setter methods. The tuple class is a basic entity and contains all the key-value pairs in the GML language. It is to be noted that even integers and floats are represented as strings. They are stored as strings in this tuple class.

EntInfo

The EntInfo class consists of the following four fields.

1. att - array of tuples
2. link - boolean
3. color - long int
4. name - string

The EntInfo class contains the properties for a single node or a link. The *link* field is true if the entity is a link and is false if it is a node. *color* specifies

the colour of the entity and *name* gives its name. The array *att* contains the list of properties in the form of key-value pairs (tuples). For the schema these contain pairs like <property_name , property_type> and for the instance of the schema pairs like <property_name , property_value>. Several methods are defined on this class. They basically add, delete and modify properties in the *att* array.

Info

This class contains the schema of the full network. It has just one field *att* that is an array of *EntInfo* classes. As each such class represents information about one node or link, their collection is the full schema. This class maintains the full schema and several methods are defined on it that manipulate the schemas. The methods are very similar to those defined on any standard collections class. There are methods to add, delete and modify entries etc.

Serialization Methods

All the above classes have three methods in common. These are as follows.

1. writeGML(file_pointer) - This function writes the data in the object in the form of a GML file and calls the same function defined in constituent objects. Thus writeGML is called recursively. The user just needs to call this function for the parent object. Then the parent will write its data and call this function for each of its child objects. As an example when *writeGML* is invoked for the *Info* object then it calls the same *writeGML* for its constituent *EntInfo* and *tuple* objects.

2. writeXML(file_pointer) - This is similar to the function described above. The only difference being that instead of writing the data in GML it is written out in XML.

3. Serialize(CArchive object) - This is an MFC function. This writes out the data as a standard windows binary. It is invoked in exactly the same manner as the earlier two functions. Instead of writing a text file it writes a binary file with the file extensions (.ntg) and (.sch). (.ntg) file extension is for the full application consisting of the schema, schema instance and drawing. (.sch) is a file just for the schema.

3.3 Drawing Editor

After building the schema it is time to draw the network and fill in the properties of its constituent nodes and links. The drawing editor is shown in figure 3.3.2.

3.3.1 Features

Figure 3.3.2 depicts the situation where five nodes have been drawn and they have been interconnected with several links. These nodes were drawn by selecting the drawing button on the toolbar and then drawing the node using the

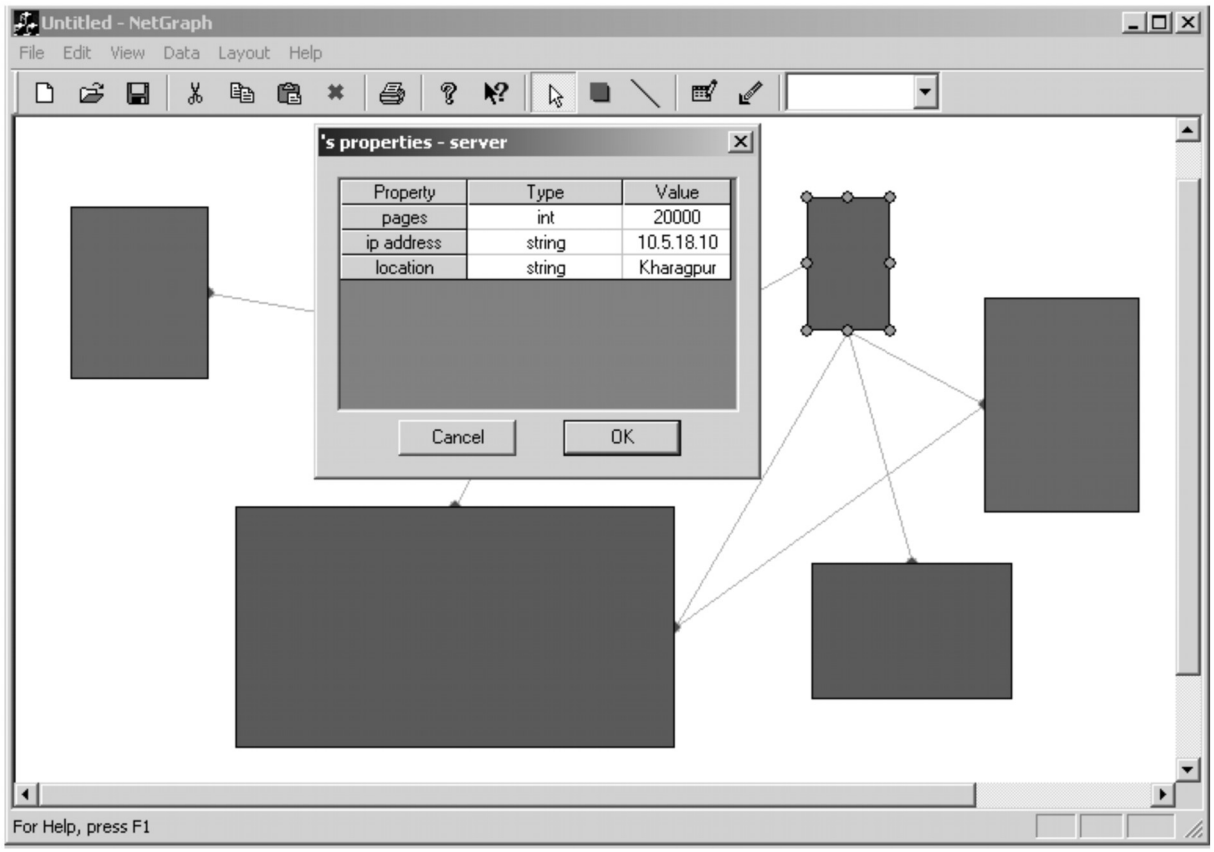

Figure 3.3.2: Drawing Editor

mouse. The links were drawn by selecting the link button on the toolbar and then drawing the link by clicking on the source and destination nodes. The link automatically found its correct orientation to minimize the number of crossings. To select the specific node or link there is a combo box in the right hand corner of the toolbar. There the user can select the specific node or link to draw.

The other buttons on the toolbar implement the standard functions of cut, copy, paste and clear. To the extreme left are the buttons that implement file open, save and create. After creating the schema the user just clicks the node or link buttons to go to node-drawing mode or link-drawing mode and keeps drawing. Beside the node button on the toolbar there is the pointer button. This doesn't draw anything but is useful in selecting an object or a group of objects. These objects can be cut or copied and pasted in a different place or in a differnt file. They can also be moved and resized. There are also several layout functions. They are accessible through the layout menu item. They can make elements of the same size, align them and space them equally. In the pointer mode it is possible to edit the properties of a node or a link by double clicking it. This has been done for the server node in the figure. A property box has popped up near it. It shows the properties that were entered during the time the schema was made. Now the user puts in the values of the properties for the corresponding node.

In this manner the drawing proceeds. The important feature of the NetGraph environment is that the user need not bother about how links are placed. He just clicks the source and the destination in link mode. NetGraph finds the best possible orientation for the link and draws the link accordingly. Whenever objects are moved or resized links change their position such that the node does not overlap with it and it has minimum length.

As has been mentioned earlier the drawing can be exported in XML and GML file formats. There are two issues worth mentioning. First, what happens if a portion of a drawing is copied from one instance of a NetGraph application to another instance? If they have the same schema or the schema of the former is a subset of that of the latter then there is no problem. It is like a cut and paste operation in the same application. But if the schema is different then there is a problem. In this case the user is prompted. He has a choice. He can either cancel the operation or he can request NetGraph to computer an intersection of the two schemas and paste the common part. If this is done then some attributes and some nodes and links might not be pasted on the latter's drawing area.

The second issue is related with compatibility. Some GML files of other simulators do not have position information. Thus, it is not possible to draw the network when such a GML file is available. So in that situation instead of discarding the GML file NetGraph guesses a drawing and draws it. This is a very simplistic drawing and nodes are just placed in a mesh. It is up to the user to beautify the drawing.

3.3.2 Implementation

Unlike the schema section this section is very heavily dependent on the MFC Document/View architecture. The document consists of the schema and the drawing and the view encapsulates the drawing area. System events are trapped by the document and user events like mouse-clicks are trapped by the view class. The major data structures used to implement the drawing functionality are as follows. The notation used to represent the prototypes of variables and methods uses MFC data structures.

ViewOb

This is an abstract superclass. It represents a generic view object. It is inherited by node and link objects. It just represents the common part. It has five member variables which are as follows:

1. String caption - This is the name of the node or link object. At the moment this is not used in NetGraph but in future versions the name of the object will appear in the drawing.
2. int num - This is a numerical id that uniquely identifies the object. It is to be noted that node and link ids are in separate spaces and they are never compared with each other. So, it is possible ot identify every drawing object by its unique id. Whenever an object is deleted or extra objects are created the ids are readjusted to form an uniformly increasing sequence. If the id of a node is changed then the data structures of the links adjacent to it are modified to reflect the change.
3. CRect rect - This is a rectangle structure. For the node this gives the top-left and bottom-right co-ordinates. For the link the first point gives the source and the second gives the destination.
4. CRect prect - When a node or link is being adjusted it is previewed with dotted lines. This rectangle maintains those co-ordinates.
5. EntInfo ent - This maintains the list of properties in the form of key-value pairs.

Thus, this object stores the temporary and permanent position of the node or link that inherits it. It also holds its unique id and a list of attributes. A couple of methods are defined on this object. They are all pure virtual. They are meant to be overridden in child classes. They are as follows:

- bool inOb(CPoint) - This function returns a boolean value indicating whether the point lies inside the object or not. In the case of a node a true value is returned if the point is within the object. In the case of a link a true value is returned if the point of clicking is within a close proximity of the link.
- bool contained(CRect) - This function returns a true value if the object is contained within the rectangle that is passed as a parameter.

- void draw() - A node or a link draws itself.
- void preview() - A node or a link draws itself in preview mode. That means that instead of solid lines the lines are drawn as dashed ones.
- void highlight() - The object highlights itself.
- void sync(Info*) - This is a synchronization function. This is useful in the cases when the schema is changed at runtime or a node is pasted in another instance of NetGraph with a different schema. Then the object needs to remove all the items in its property lists which are not part of the target schema. This function does this synchronization. This is also implemented in a recursive fashion. Calling the *sync* method of the parent leads to the invoking of the *sync* method of all its children.

NodeOb

This class inherits from *ViewOb* and does not have any extra member variables. It provides implementations for the abstract functions. Along with the abstract functions it provides implementations for the three standard function *writeGML*, *writeXML* and *Serialize*.

LinkOb

This class inherits from *LinkOb* and overrides all its pure virtual functions. It does not have any extra method of its own. But it has two extra member variables. They are two integers *source* and *target*. They are the ids of the source and target nodes. Every link has information about the nodes that it connects. Whenever those nodes change position or are deleted the link needs to modify its properties. If any of the end nodes are deleted then the link is deleted. If the destination nodes change position or are resized then the link's *preview* field is updated with this new information. The link draws itself in preview mode as long as the user has the left mouse button pressed. When it is released the nodes draw themselves at the released point and set their new locations as their permanent locations. Then the links whose end nodes have changed positions calculate their new positions with the following optimization criteria in mind. They try to avoid an overlay with their end nodes. They also try to minimize the length. After getting the new position they set it as their permanent position. Thus, in this manner nodes and links interact to maintain consistency in the drawing.

ObjGroup

This contains an array of nodes and an array of links representing a portion of the drawing. This is a very useful data structure. The full document can be represented as an *ObjGroup*. Even parts of it like the current selection and the data to be sent to or from the clipboard are *ObjGroups*. This data structure efficiently encapsulates a group of nodes and links and is passed from function to function for processing.

Several methods are defined for this class. Some of them are data manipulation methods that add, delete or check the existence of a node or a link. The other methods defined on this are as follows:

- void draw() - All the objects draw themselves.
- void highlight() - All the objects highlight themselves.
- void move() - This causes all the nodes and links to move to the new location. This is a very simple method because all this does is that it sets the *rect* member equal to the *prect* member. This basically means that it sets the permanent address equal to the temporary address. This is simple for nodes. After that links must calculate their new positions and set themselves.
- void reLocate() - This prunes of all the links that have either one or both their end nodes missing. It relocates the rest to the top left corner of the virtual co-ordinate space.

Selection

This represents the set of objects that have been currently selected. It just has one member variable. It is an *ObjGroup* containing the nodes and links that have been selected. There are some extra methods defined on this group of objects. They are outlined below.

- void moved(CPoint) - This means that the pointer has moved to a new position and the objects in the group must be relocated to the new position. As long as the mouse is down the objects in the selection are just previewed. When the mouse is released their permanent position is set equal to their temporary locations.
- void align(int code) - Depending upon the type of alignment requested the objects in the selection are aligned. This means that their new positions are calculated as well as the links are adjusted to reflect the change in the position of the nodes. It is assumed that links are passive and are completely dependent on their end nodes. They cannot be moved or resized. They find their own position.
- void spaceEvenly(int code) - This method causes the nodes in the selection to space themselves either vertically, horizontally or in both ways.
- void makeSameSize(int code) - The nodes in the selection are made to have the same size. Depending upon the code they set their heights, widths or both equal to that of the average values of the whole group.

3.4 Scope of the Tool

The tool has an object oriented design that is completely embedded within the Document/View architecture of the MFC framework. Thus, it is easily

extensible and customisable. If later it is desired to add some extra features then they can be added without much effort. The MFC classes that are not described in the thesis just consist of event handlers that handle the screen events and update the data structures. In the future we hope that more ActiveX, COM and .Net controls will be added to provide a rich development environment.

The tool has been made compatible with standard graph markup languages like GML and XML. The output can be fed to parsers and the output of other programs can be fed as an input to NetGraph. NetGraph can also print the drawing to standard printers or to virtual printers. There is also a facility to export the drawing as a bitmap image. Once the bitmap image is obtained it can be converted to any other image format and it can become an input to technical documents that show the network topology.

Chapter 4

GRAPES - request generation scheme

4.1 Introduction

In this section we focus on generation of web workloads required for simulation studies of WAN traffic. Several analytic models and invariants have been proposed based on extensive studies of WAN traffic [2, 1]. Some of these analytic models are used by web workload generators like SURGE [3] and ProWGen [6]. They generate artificial workloads whose statistical properties conform to the analytic models and obey certain invariants as stated in [2]. SURGE is a very popular workload generator for synthesizing the workload generated by an individual user. ProWGen has been used to investigate the sensitivity of proxy cache replacement policies to selected workload characteristics [6]. SPECweb99 and WebBench are popular benchmarking software for web servers and proxies.

A workload generator generates requests for web pages and times them. It deals with four main concerns. These are (i) generating web request streams obeying the empirically observed popularity and temporal locality distributions, (ii) generating embedded requests, (iii) accounting for the correlations between file size and file popularity and (iv) timing the requests. Generation of embedded requests and timing of requests can be done on the fly as requests are generated [3]. Information about embedded pages can be stored apriori and processed at run time [3]. Normally a table of embedded references is maintained. Whenever there is a request for the main page requests for the embedded objects are inserted into the request stream. Inter-request times are calculated from a probability distribution or by any other scheme resulting in a constant overhead per request. Correlations between file sizes and file popularity can also be taken care of as in SURGE. We observe that the most expensive operation is that of combining temporal locality of web page access with page popularity distributions. The other concerns in the generation of request streams are settled; they are either handled in a pre-processing step or they add only a constant overhead to the step that generates the subsequent request. Therefore, we concentrate only on the problem of generating a request stream with the desired properties of temporal locality and page popularity and we keep the rest out of the scope

of this paper. Both SURGE and ProWGen explicitly use stacks and certain stack-distance distributions to realize temporal locality of web page accesses. The time complexity of this step in the SURGE generator is $O(mn)$ in the worst case, where m is the number of pages and n is the number of requests generated. Similarly, the time complexity of this step in ProWGen is $O(sn)$ where s is the size of the stack and n is the number of requests. The plots of running times versus the number of generated requests (see Figure 4.6.5) show a super linear trend. This trend is due to the stack manipulation step which is executed for every newly generated request. In this paper we propose an efficient algorithm GRAPES (Scalable Efficient Parallel Artificial Request Generator), for generating web page request streams satisfying the empirically observed Zipf page popularity distribution and a temporal locality property in the form of log-normal stack-distance distribution. We avoid explicit use of the stack as done in previous generators like SURGE and ProWGen. At the outset we compute a set of coefficients representing a discrete probability distribution that acts as a *signature* of the desired stack-distance distribution. This discrete probability distribution is computed apriori and used for generating request streams of any length in time proportional to the number of generated requests; the precomputed probability distribution essentially models stack-distance distribution and enables us to do away with the stack. Our algorithm GRAPES can replace the corresponding algorithm in the SURGE generator keeping the rest of SURGE intact to provide a much faster package for generating web workloads. In contrast to known workload generators that certainly run in super linear time, the use of our algorithm in a classical workload generator would allow it to generate requests in linear time.

Our work is motivated by the need to simulate a WAN environment having millions of pages and web-page requests. For this purpose we require a scalable and efficient web workload generator. Our algorithm runs an order of magnitude faster than the corresponding algorithms in existing simulators/workload generators and hence it is scalable for use in very large simulations. The time complexity of our algorithm is independent of the number of pages whereas the time complexities of the earlier workload generators in [3, 6] depend on the number of pages. For a large number of pages, our algorithm can give a tremendous performance advantage. However, the resources of a single processor machine might prove to be inadequate for running very large simulations. We may need to resort to using multiprocessor machines for performing very large simulations. The other stack based algorithms do not seem to admit a parallel solution because of the stack manipulation step. In contrast, our algorithm admits a very simple parallel solution that runs in $O(n/p)$ time in the CRCW PRAM model with $p < m/\log m$ processors, where n is the number of requests generated and m is the number of pages. Thus, our algorithm can be used for generating request streams for very large simulations. Along with the observed superior speed, scalability and parallelizability, the plots of the statistical properties match very well with empirically observed distributions (see section 4.6).

The novelty of our approach is that we do away with the explicit use of the stack for generating each subsequent request. Instead, we develop a signature of

a predetermined number (say 800) of coefficients representing a discrete probability distribution from the distribution obeyed by stack-distances (see section 4.3). This signature is used in our request generation algorithm presented in section 4.4. Since this signature is a property of log-normal stack-distance distributions, it is like a table which can be generated once and stored away so that any number of request streams of arbitrarily large lengths can be generated using this table. We quantitatively compare the performance of our algorithm with respect to existing request generators and analytical models in section 4.6. In section 4.5 we parallelize our request generation algorithm. We start by summarizing the previous work done in this area.

4.2 Summary of Previous Work

Several studies have been carried out on web request streams and their statistical properties. Our paper will concentrate chiefly on two of them - page popularity and temporal locality.

Page Popularity Page popularity refers to the relative number of requests made to individual pages. Several studies have been carried out on popularity distributions. Page popularity has been observed to follow Zipf's law (see [5]). Zipf's law states that the popularity of the i^{th} most popular page varies inversely with i. Stated mathematically the probability mass function p is given by:

$$p(i) = \frac{\Omega}{i}$$

Temporal Locality This property refers to the probability that if a page has been requested, it will be requested again in the future. To characterize this the stack-distance model has been proposed (see [3, 1]).

The stack-distance model is as follows. Let the pages be organized in the form of a stack $< p_1, p_2, \ldots, p_n >$ with p_1 at the top of the stack. Suppose the next request is for the page p_i. Then the stack-distance d for this request is i. Stack-distance measures the depth within the stack where the next request is found. Page i is moved to the top of the stack. The new stack configuration is as follows $< p_i, p_1, p_2, \ldots p_{i-1}, p_{i+1}, \ldots, p_n >$. Thus given the configuration of the initial stack, a stream of stack-distances is sufficient to describe the request stream. The information contained by both the representations is the same. The stack-distance method is a better representation of the request stream as it is an ordered sequence of numbers and not pages. Stack-distance distribution serves to characterize temporal locality. A small stack-distance implies more temporal locality. The stack-distance has been observed to follow the log-normal distribution (see [3, 1]). The log-normal distribution is given by:

$$lgn(x) = \frac{1}{2.3x} * \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(log(x)-\mu)^2}{2\sigma^2}}$$

The logs are to the base 10.

> The mean and variance of the log-normal distribution for stack-distances have been calculated for various traces (see [1]). The mean values are shown to lie between 1.5 and 2.0 and the standard deviation between 0.8 and 0.9.

Several web request stream generators have been built which try to incorporate the above mentioned properties. Two of these generators are SURGE and ProWGen. SURGE starts with a stack of web pages. In every iteration it generates a stack-distance d obeying the log-normal distribution. The page at depth d in the stack is removed and pushed to the stack top. This page is added to the request stream. To ensure homogeneity and adherence to the Zipf distribution of page popularity, there are some extra steps in the algorithm. After generating the stack-distance a window is defined around the requested page within the stack. Pages within this window are assigned weights depending upon the number of requests left. The page with the maximum weight is moved to the top of the stack and added to the request stream. Let n be the number of requests to be generated and m be the number of pages. Since for each request generated the stack of size m is updated, the worst case complexity is $O(mn)$.

ProWGen is another web request stream generator. ProWGen incorporates only those characteristics that are relevant to web caching. In addition, ProWGen models only the aggregate client workloads seen by a typical proxy server, rather than individual clients. ProWGen also uses a LRU stack to model the temporal locality. But ProWGen generates a stack-distance from a uniform distribution rather than the log-normal distribution used in SURGE. Let n be the number of requests to be generated and s be the size of the stack. As in the case with SURGE, the asymptotic worst case complexity of this algorithm is $O(ns)$. While the stack size of SURGE was equal to the number of pages, the stack size of ProWGen is meant to be set by the user.

Due to the explicit use of the stack, both the above algorithms have time complexity proportional to the product of the stack size and the length of the generated request stream. Our algorithm GRAPES in section 4.4 does away with the stack and uses the signature developed in the next section to achieve an asymptotic worst case time complexity that is linear in the number of requests and independent of the number of pages. In the next section we show how to generate the short signature used in section 4.4.

4.3 Generating the Signature

As observed in earlier workload generators like SURGE [3] and ProWGen [6], the step implementing temporal locality uses the inherently expensive stack manipulation step. The stack is used to incorporate temporal locality of web page references by realizing the empirically observed and hitherto widely accepted log-normal stack-distance distribution [1]. SURGE uses log-normal distributions and ProWGen uses an uniform distribution for stack-distances. However, the stack manipulation step necessary for generating every subsequent page request is expensive and is proportional to the size of the stack in the worst case.

Larger stack sizes would be necessitated by the increase in the number of web pages. Moreover, the number of generated requests in meaningful simulations would have to rise in proportion to the number of pages. The workload generator would therefore suffer from a quadratic time complexity with respect to the number of generated requests (see Figure 4.6.5). In this section we show how to get rid of stack operations in the generation of web page request streams. We generate a short but useful *signature* apriori from the log-normal distribution for stack-distances. This signature is then used to generate web page request streams of arbitrary lengths for an arbitrary number of web pages. The signature generation step uses a novel dynamic programming technique to generate the first k *first-return probabilities* for a page. The *first-return probability* is the probability with which the distance between two consecutive requests to the same page takes a certain value. In particular, we define the term *inter-request distance* as the number of requests between two consecutive requests to the same page plus 1. We propose to obtain the first k first-return probabilities for an arbitrary page in an artificial request stream as the signature in this section.

Now we develop the procedure for computing the discrete distribution of the first k first-return probabilities for pages. We found that it suffices to set k equal to 800, irrespective of the number of requests we need to generate. We assume that the distributions of all pages are i.i.d. This was observed by Williamson et al. (see [2]). So, it suffices to concentrate on one page and study how its request instances are distributed in the long run, at steady state. We assume that the number of pages is finite, however large. The first-return probabilities computed apriori in this section are used to generate web page request streams of arbitrary lengths using the algorithm of the next section in time proportional to the number of generated requests. The inter-request distance distribution generated in this section is used for generating long request streams with page requests following the same distribution as derived here. We claim that the generated request stream closely obeys the log-normal stack-distance distribution and a few additional desirable properties. This claim is substantiated in section 4.6.

In order to compute the signature for the stack-distance distribution, let us consider a stack of m pages where $m > k$. Let us consider the situation when the page starts from the top of the stack to begin with. The page can move down the stack by at most one position for each generated page request. Finally the page jumps to the top of the stack. The movement of the page from depth i is governed by the next stack-distance generated (by say, the log-normal distribution). If the distance generated is greater than i, the page shifts to depth $i + 1$. If distance i is generated, page i returns to the top of the stack. Otherwise, the page remains at position i. Each of these three cases are executed with finite probability. Our main aim is to find the probability that the page returns to the top after j steps, $1 \leq j \leq k$ without returning to the stack top in less than j steps. This would be the j^{th} first-return probability, f_j. Returning to the top of the stack for the first time in $j+1$ steps is equivalent to reaching some depth i in the stack in $j \geq i - 1$ steps (without returning to the top) with probability say $q_{i,j}$ and, then making one jump to the top of the stack

with probability p_i. Note that p_i is the normalized [1] log-normal probability for stack-distance i. Hence,

$$f_{j+1} = \sum_{i=1}^{j+1} q_{i,j}\, p_i$$

We do not need to add more than $j+1$ terms because the stack-distance model prevents a page from visiting positions below depth $j+1$ in j steps if the page starts at the top of the stack. In particular, note that f_1 is identical to p_1.

Now we consider $q_{i,j}$, $1 \leq i, j \leq k$, the probability with which the page reaches stack depth i in j steps, given that the page started at the top of the stack and never reached the top of the stack in l steps where $0 < l \leq j$. Since the page starts at the top of the stack, we set $q_{1,0} = 1$ and $q_{i,0} = 0$, $1 < i \leq k$. Surely, the page must move down to depth 2 in the first step, implying $q_{1,1} = 0$. It must also not return to the top in j steps, rather it should keep moving down, at most one step each time a page request is generated– so we fix $q_{1,j} = 0$, $1 \leq j \leq k$.

Now consider how the page reaches depth i in j steps. Since the page can move at most one step down each time, it could reach depth i after $j \geq i-1$ steps in only two ways (i) remaining at depth i after it reaches depth i in $j-1$ steps with probability $q_{i,j-1}$ or (ii) moving down from depth $i-1$ to depth i after reaching depth $i-1$ in $j-1$ steps with probability $q_{i-1.j-1}$. The page can transit from depth i to the same depth with probability $a_i = \sum_{j=1}^{i-1} p_j$. The probability that the page moves down one step is $b_i = 1 - a_i$. So, we can write $q_{i,j} = q_{i-1,j-1}b_i + q_{i,j-1}a_i, 2 \leq i \leq k$

We summarize the above discussion in the following theorem.

THEOREM 4.3.1 Let $q_{i,j}$ be the probability that a page starts at the top of the stack and moves to stack depth i in j steps without reaching the stack top in between, where $1 \leq i \leq k$ and $0 \leq j \leq k-1$. Then,

$$q_{1,0} = 1$$

$$q_{i,0} = 0 \ , 1 < i \leq k$$

$$q_{1,j} = 0 \ , 1 \leq j \leq k-1$$

$$q_{i,j} = q_{i-1,j-1}b_i + q_{i,j-1}a_i \ , 2 \leq i \leq k \ , 1 \leq j \leq k-1$$

The computation of $q_{i,j}$'s and hence f_j's can be done using dynamic programming in $O(k^2)$ steps. Thus we can determine the small signature consisting of the first k first-return probabilities. Note that the k first return probabilities may not add up to unity. The deficiency is distributed over the next k pages to obtain a tail in the discrete probability distribution with a total of $2k$ coefficients. We observe that if the number of pages is sufficiently large then

[1] The log-normal probabilities for the m stack-distances are added up and normalized to obtain the values of p_i, $1 \leq i \leq m$, so that $\sum_{i=1}^{m} p_i = 1$.

these coefficients are independent of the number of pages. This claim is substantiated by exhaustive experimentation. Let us consider an example. Let the log-normal parameters μ and σ be 1.75 and 0.85 respectively. We generated 800 coefficients for a large number of pages. It was observed that after 5000 pages the maximum difference between corresponding coefficients did not exceed 0.001. Similar results were obtained with different log-normal parameters. Thus it suffices to generate the coefficients for n_0 pages where n_0 is sufficiently large and use it for any $n \geq n_0$ pages.

Note that the recursive definition did not assume the log-normal stack-distance distribution at any stage. Our scheme therefore permits any statistical distribution for stack-distances.

4.4 Request Generation Algorithm

We state our request generation algorithm GRAPES in this section. The inputs to the algorithm are the number n of requests to be generated and, the number m of pages for which we require to generate requests. The pages are assumed to be numbered from 1 through m. GRAPES fills up an array arr with page requests. Each position of the array contains an integer i between 1 and m, representing a request for the i^{th} page.

GRAPES aims to generate a request stream with the following properties:

1. Zipf page popularity distribution (see section 4.2).
2. Inter-request distance distribution for each page equal to that derived in section 4.3.
3. Uniformly and randomly distributed page requests.

The request stream's page popularity distribution follows Zipf's law. We claim that log-normal stack-distance distribution is achieved for generated requests streams. This claim is further substantiated by the plots in section 4.6. The third condition implies that the request stream generated is not predictable and all the requests to a page are not temporally close by. They are sufficiently spread out in time. Indeed, the techniques we use to achieve the third condition above also have a remarkable off-shoot in enabling us to design a very simple parallel implementation of our algorithm GRAPES (see section 4.5).

Now we outline our algorithm. The algorithm applies the Zipf distribution to determine the number of requests to be generated for each page. It repeatedly selects a random page p and generates a random number of requests for the selected page p. For each page, it maintains the remaining number of requests to be generated.

Now consider the generation of a set of requests for a page p. These requests are placed at appropriate positions in the request array following the inter-request distance distribution derived in section 4.3. While attempting to place a request for p at a position x in the array, we may encounter a clash; a request for another page q might have been placed at position x already. This clash is resolved by placing a request for p at x and relocating the request to the

other page q to another position in the array. The relocation of q is done in accordance with the second condition, not violating the inter-request distance distribution for page q. The position where we wish to relocate q may also be occupied by another page request, allocated earlier to say page r. In this case we repeat the relocation step (as done for page p earlier); we force page q in the location and evict r to relocate it. This chain of relocations due to clashes is continued for a constant number of steps, thereby limiting the time complexity of the algorithm at the minor cost of not being able to relocate the last evicted page in any such chain of relocations.

Next, we delve into the minutiae of the algorithm. At the outset each position of the array *arr* is initialized to -1. In each iteration of the main loop we generate a random page p and a random number *numRequests* of requests for the page p. These requests are added to the array by the function `addToArray` stated below. This function takes the page p and *numRequests* as inputs.

Line 1 initializes a static variable *pos* that remembers the position of the last request placed in the array. The main for loop iterates over all the requests to be generated. If the array is ninety percent full then the function `isMostlyFull` returns true. Then lines 4-12 find the next free position in the array within a window of size FREE_SEARCH_LIMIT. If such a position is found then the request is placed in the array; otherwise, the request is not placed and ignored. If the array is less than ninety percent full then the request is tried to be placed at the position *pos*. If this position is occupied by a request for another page then the routine `force` is invoked. Finally the value of *pos* is updated. A random number *idist* is generated that follows the inter-request distance distribution derived in section 4.3. This random number *idist* is added to *pos* giving the position where the next request for page p is intended to be placed. This is in accordance with condition 2.

```
addToArray(p,numRequests)
1    static int pos = 0
2    for (i=1;i <=numRequests; i++)
3        if(isMostlyFull())
4            flag = false
5            k = 0
6            while (k<FREE_SEARCH_LIMIT)
7                pos = (pos+1) % n
8                if(free(arr[pos]))
9                    flag = true
10                   break
11               k++
12           if(flag) arr[pos] = p
13       else
14           if(free(arr[pos])) then arr[pos] = p
15           else force(pos,p)
16       idist = getInterRequestDistance()
```

```
17        pos += idist
18        pos = pos % n
```

The function `force` takes as parameters the clashed position *origpos* and the page p that could not be placed at *origpos*. Lines 1-2 place p at *origpos* and evict the previous request to another page. The number of this page is stored in the variable t. Line 7 finds a new position *pos* for t. Then t is placed at *pos* and the previous request at *pos* is evicted and placed in t. This procedure continues until an empty position is found or until the number of iterations is greater than or equal to $ITER\_LIMIT$.

```
force(origpos,p)
     /* evict the existing request */
1    t = arr[origpos]
2    arr[origpos] = p
3    limit = 0
4    pos = origpos
5    while(limit < ITER_LIMIT)
6        limit ++
7        pos = getNewPos(pos,t)
8        swap(arr[pos],t)
9        if(t == -1) return
```

The function `getNewPos` takes as its parameters the position *pos* and the evicted request for the page t. This function tries to relocate t to a new position such that the distribution of inter-request distances is maintained. Line 1 tries to determine the position *lower* of another request for page t in the array within a reasonable sized window($SPAN$) around *pos* such that:

$$(arr[lower] = t) \wedge (|pos - lower| < SPAN) \wedge (\forall i \in (lower, pos)\, arr[i] \neq t)$$

Similarly *upper* is the position of another request for page t such that :

$$(arr[upper] = t) \wedge (|pos - upper| < SPAN) \wedge (\forall i \in (pos, upper)\, arr[i] \neq t)$$

First we consider the case where we discover valid values for *lower* and *upper*. Since t is being evicted from position *pos*, a suitable new position for t would be *newpos* as determined in Line 6; this position has the same set of distances from *lower* and *upper* as *pos* with the distances simply swapped. In other words, $newpos - lower = upper - pos$ and $upper - newpos = pos - lower$. This crucial distance swapping step helps preserve the inter-request distance distribution. This maintenance of the inter-request distance distribution ensures log-normal stack-distance distribution for the generated request stream. Lines 7-8 handle the special cases where $pos = newpos$.

Now consider the case where either of *lower* or *upper* does not exist. In this case a new position is searched in a smaller window of size WIN_SIZE around

pos. The function `searchWithinWindow` returns a random empty position in such a window and, if such a position does not exist then it returns a random filled position.

```
    getNewPos(pos,t)
1       lower = getNextLowerPage(pos,SPAN)
2       upper = getNextUpperPage(pos,SPAN)
3       if(!valid(lower) || !valid(upper))
4           newpos = searchWithinWindow(pos,WIN\_SIZE)
5       else
            /* relocate to a new position */
6           newpos = lower + upper - pos

            /* correction for the midpoint */
7           if(newpos == pos) newpos ++ ;
8           if(newpos == upper) newpos = (newpos + 1)%n
9       return newpos
```

In this manner GRAPES generates requests for pages. Now, we discuss some intricacies of the algorithm. Due to the random choice of the next page to be allocated in the array and the wraparound strategy adopted in the function `addToArray`, a request for a page might be allocated in an area of the array where requests for the same page might already have been allocated earlier. This would violate the inter-request distance distribution. We have chosen to ignore this as such aberrations do not change the nature of the final stack-distance plot as seen in section 4.6. Note that the function `force` and the linear search phase in the function `addToArray` might sometimes fail to add a request. The percentage of requests lost depends upon the constants used in the algorithm. This leads to a tradeoff between the running time and the loss percentage. We propose heuristics that ensure a fairly fast running time and less than 0.1% loss. These heuristics are derived from exhaustive experimentation and hold for a varied number of input sizes and log-normal parameters. To state these heuristics we need to define the mean λ of the inter-request distance distribution. This mean turns out to be between 100-150 when the signature distribution determined in section 4.3 has 800 coefficients (first-return probabilities) and the log-normal parameters are within the ranges specified in section 4.2. The choice of the constants realizing our heuristic are as follows.

- FREE_SEARCH_LIMIT = 5λ
- ITER_LIMIT = $\frac{\lambda}{3}$
- SPAN = λ
- WIN_SIZE = $\frac{\lambda}{10}$

Now we are in a position to calculate the asymptotic complexity of the algorithm. The function `getNewPos` takes $O(1)$ time. `force` invokes `getNewPos` at most ITER_LIMIT times. Thus `force` also runs in $O(1)$ time. Now for each

request added `addToArray` either searches at most FREE_SEARCH_LIMIT positions or calls the `force` function. Both take $O(1)$ time. Since there is a constant overhead per each request generated, the worst case asymptotic running time complexity is $O(n)$ where n is the number of requests generated.

4.5 Parallel Request Generation Algorithm

The algorithm presented in the previous section permits a simple parallel implementation. We state the parallel version in this section. The key to the parallel implementation is in the probabilistic approach; randomly select a page and randomly choose a number of requests to be generated for the selected page.

We assign pages to processors, partition the requests to be generated and give each processor its share. Then the processors can start at different positions in the array and just place the requests as if they were isolated. Each processor would independently run the sequential algorithm. While explaining this idea we assume the PRAM model. The details about this model can be found in references [9, 12].

Let us assume that we want to generate n requests for m pages with p processors. Without loss of generality let us assume that n and m are multiples of p and $n > m$. Let us assume that the number of processors $p < m/\lg(m)$ which will be the case in all practical scenarios. Let us consider a priority based CRCW model. This condition will be relaxed later. At the outset each processor obtains the list of pages and the number of requests for each page it will generate. This can be done as follows.

- Create an m element array whose ith element has the number of requests to be generated for the ith page. This is calculated from the Zipf distribution (see section 4.2). This takes $O(\frac{m}{p})$ time and can be executed on an EREW PRAM.
- Calculate the prefix sum in $O(\frac{m}{p})$ time. This can also be executed on an EREW PRAM.
- Each processor with id j searches for the position of the $((j-1)*\frac{n}{p})$th request and the $(j*\frac{n}{p}-1)$th request in the array. This can be accomplished in $O(lg(m))$ time on a CREW PRAM. Now each processor knows the list of pages and the number of requests for each page it has to generate. The total number of requests each processor generates is equal to $\frac{n}{p}$.

Subsequently each processor with id j starts at the $((j-1)*\frac{n}{p})$th location in the request array and runs the sequential algorithm for the subset of requests that it has to generate. If more than two processors want to write to the same location then the one with highest processor id succeeds. The rest of the processors try to relocate the request that they couldn't place in the array (see function `getNewPos` in section 4.4). In this manner a request stream is generated.

A special situation might arise when a request for page p gets evicted by one processor and the next request to p in the array gets evicted by another

processor. This will cause the method `getNewPos` to function anomalously and will disturb the inter-request distance distribution. By exhaustive experimentation and probabilistic analysis of some representative cases, the probability of such an error was found to be very low. Hence, we chose to ignore it.

Thus, the request generation process takes $O(\frac{n}{p})$ time because the sequential algorithm is linear in the number of requests. Summing up all the running times the net worst case asymptotic complexity becomes $O(\frac{n}{p})$. Now, let us try to generalize our result to CREW and EREW PRAMS. We would like to refer to the following result from Jaja's book (see Corollary 10.1 in [12]).

LEMMA 4.5.1 Given an algorithm that can be implemented to run in time T on a p processor priority CRCW PRAM, this algorithm can be implemented on a p processor EREW PRAM to run in $O(T \lg(p))$ time.

The scheme adopted in this book can be used to transform our algorithm to run on CREW and EREW PRAMs. Thus the worst case time complexity on such architectures is $O(\frac{n}{p} * \lg(p))$.

4.6 Results

GRAPES was coded in C++ and request streams were generated for different numbers of pages and log-normal parameters. We compare the performance of GRAPES with two popular workload generators SURGE and ProWGen. We compare the speeds of execution and the stack-distance distributions for different numbers of requests. To compare these three generators we have made certain assumptions.

1. Since GRAPES takes into consideration only page popularity and temporal locality in generating request streams, we have considered only the same subsystems in the other generators.

2. The number of requests generated in a single run of these algorithms must scale with the number of pages. We observed that setting the number of requests to ten times the number of pages allows us to generate request streams with the desired statistical properties.

3. In the case of ProWGen the stack size is meant to be set by the user whereas in SURGE it is always equal to the number of pages. For ProWGen the prescribed value of stack size is 1000. We observe that the stack size should scale with the number of pages and requests. This is necessary to meaningfully realize temporal locality. Hence, we vary the stack size of ProWGen proportionately with the number of pages. We make the stack size equal to 0.02 times the number of pages.

We have tried to model two characteristics of request streams (i) Zipf page popularity and (ii) log-normal stack-distance distribution. The former is explicitly guaranteed by the algorithm (see section 4.4). The plots in figures 4.6.1-4.6.4 depict the probability distributions of stack-distances for two different log-normal parameters and numbers of pages. The ideal distribution and

the values obtained by the SURGE generator are also plotted for the sake of comparison. Here the number of requests generated is ten times the number of pages as assumed above. The plots show that the stack-distance distribution generated by our algorithm matches very closely with the ideal distribution and that generated by the popular workload generator SURGE.

Next we come to the performance of GRAPES. GRAPES was run on a variety of machines. We present the results for Intel P-III 866 MHz running RedHat Linux 7.1. The program was compiled under g++ with maximum optimization.

pages	requests	time (sec)
5,000	50,000	0.110
10,000	100,000	0.280
25,000	250,000	0.730
50,000	500,000	1.148
100,000	1,000,000	3.05

The execution time grows linearly with the number of requests and is independent of the number of pages. This makes the algorithm pretty scalable. It can generate a million requests just in 3.05 seconds. We compare the performance of GRAPES with two popular workload generators SURGE and ProWGen. The values are plotted in figure 4.6.5. We plot the execution time as a function of the number of requests generated. The number of pages are one-tenth the number of requests.

The performance plots clearly indicate the super-linear time complexity of the algorithms used by SURGE and ProWGen. In contrast the execution time of GRAPES has a linear time complexity. This leads to a tremendous performance advantage when the number of pages and requests are large. For a million requests GRAPES is about 25 times faster than both of them. Since GRAPES is independent of the number of pages it is an order of magnitude faster than other popular workload generators for big simulations. Figures 4.6.1-4.6.5 prove that GRAPES produces request streams having the desired statistical properties and has a significantly faster execution time as compared to others.

4.7 Conclusion

We have presented an algorithm GRAPES for generating web request streams that obey Zipf page popularity distribution and log-normal stack-distance distribution. Our algorithm is based on analytical models proposed by researchers who have meticulously observed web traffic statistics. The request streams generated therefore obey empirically observed distributions of page popularity and temporal locality. The algorithm guarantees Zipf distribution for page popularity. The plots in section 4.6 show that the stack-distance distribution generated by GRAPES closely matches the ideal log-normal distribution as well as the stack-distance distribution generated by SURGE.

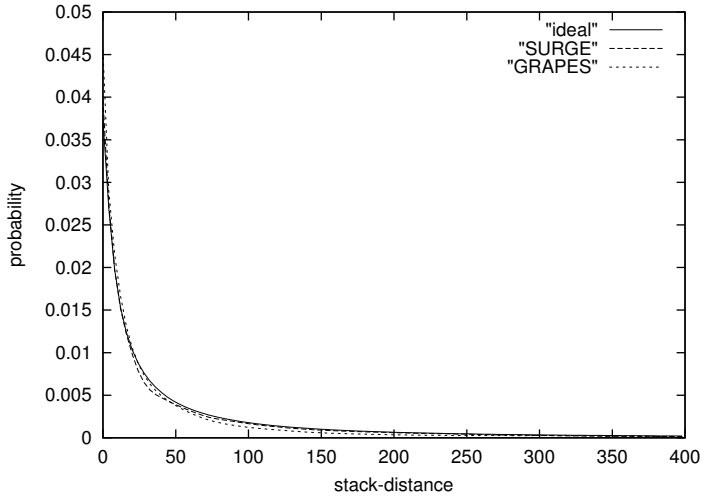

Figure 4.6.1: 10,000 pages ($\mu = 1.5$ and $\sigma = 0.8$)

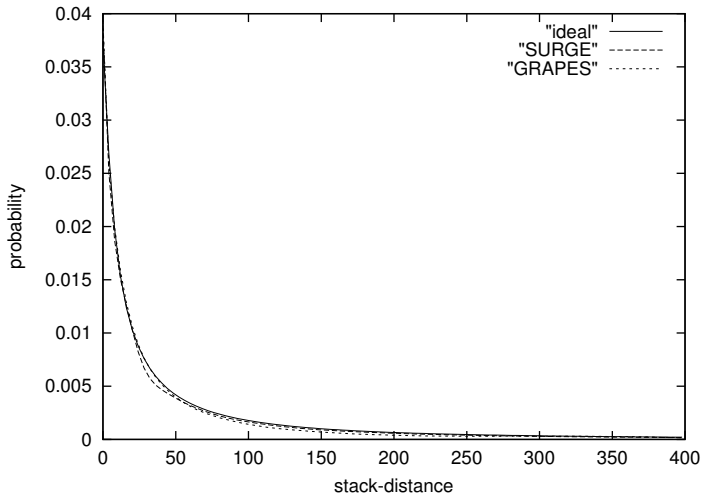

Figure 4.6.2: 50,000 pages ($\mu = 1.5$ and $\sigma = 0.8$)

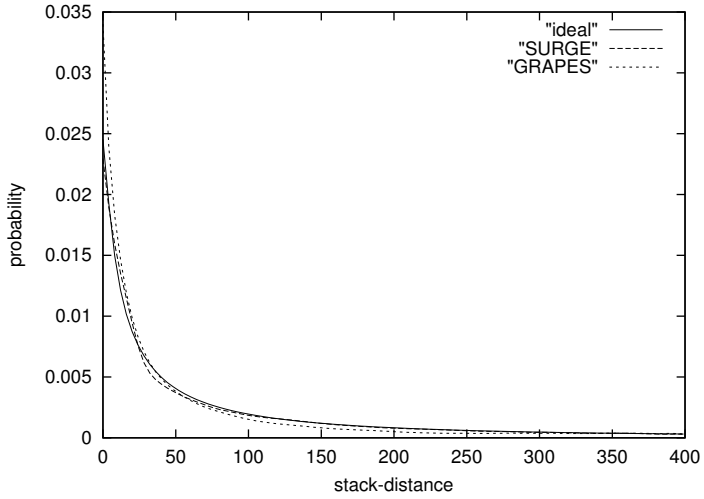

Figure 4.6.3: 10,000 pages ($\mu = 1.75$ and $\sigma = 0.85$)

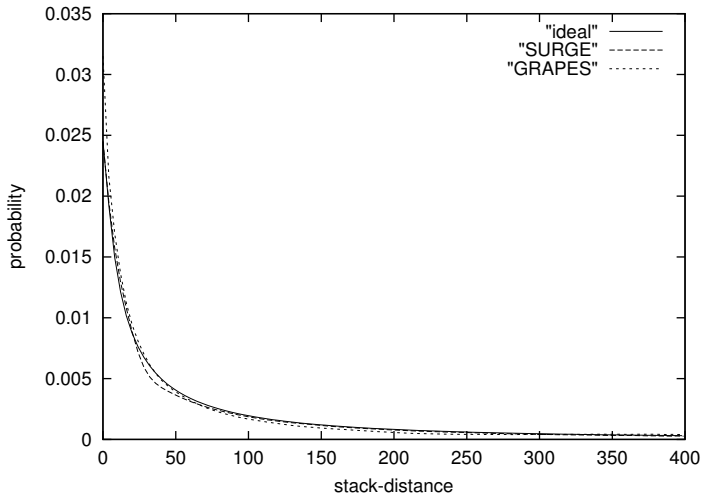

Figure 4.6.4: 50,000 pages ($\mu = 1.75$ and $\sigma = 0.85$)

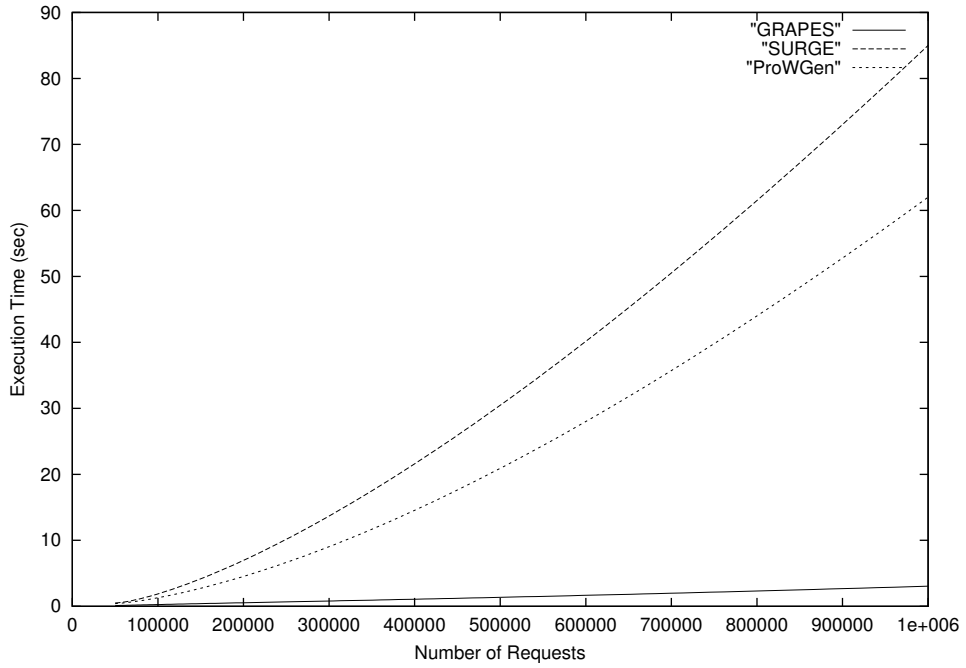

Figure 4.6.5: Performance of different request stream generators

We have used a novel approach to generate web request streams; instead of using the stack (as in earlier workload generators) we have demonstrated how to generate a set of coefficients apriori representing stack-distance distribution so that a request stream can be generated in time proportional to the number of requests, irrespective of the number of web pages being considered. Our algorithm does not assume the log-normal stack-distance distribution. Any distribution for stack-distance can be used in our algorithm; the signature must be generated apriori for the particular distribution being used.

GRAPES is a general Monte Carlo scheme for generating request streams given a stack-distance distribution. The randomization step makes the request stream unpredictable and ensures that requests to popular pages are fairly well distributed. The randomization in the algorithm has made way for a very simple parallel implementation. The parallel algorithm can be made to run on any system that can simulate or run algorithms designed for the abstract shared-memory PRAM model [12, 9].

The plots in section 4.6 show the superior performance of GRAPES as compared to other popular workload generators. GRAPES has a worst case linear time complexity as compared to the super-linear time complexity of other generators. This results in a speedup of an order of magnitude for a large number of pages and requests. These performance figures attest the suitability of GRAPES for use in large scale WAN traffic simulations.

Chapter 5

Conclusion

This thesis described the architecture of a wide area network simulator and related utilities. The simulator SWAN was coded, debugged and tested through the course of the project. Several shortcomings of other simulators were taken care of and all efforts were made to make a generic, flexible and customisable object oriented architecture. As a result the SWAN simulator can easily change its behaviour by plugging some new classes and removing the old ones. A very consistent and robust interface has been defined such that implementers can write their own classes and incorporate it into the existing framework. Provisions have also been made to incorporate the results and sub components of other simulators and workload generators.

During the course of the project two other related problems were solved that arose during the construction of the SWAN simulator. The first one was the NetGraph tool that facilitates drawing network diagrams and exporting them in various formats. This was developed in MS Visual C++. It features a very sophisticated development environment and various tools to export and import data in various formats.

Secondly, a new web request generation scheme called GRAPES was developed. This is the research contribution of the project. It combines two seemingly unrelated distributions and generates web requests conforming to both of them. There are tremendous performance advantages as compared to other web workload generators. A parallel algorithm for the PRAM model has also been presented that is work efficient.

To summarize, the project has created an environment for running large scale simulations and coducting experiments on web traffic. Associated utilites shall aid the implementer in carrying out the studies. The work done during the course of this project represents a humble beginning of a long sequence of research activities that will be carried out on WAN traffic in the future.

Chapter 6

Scope for Further Work

There were several problems that arose during the course of the project that could not be solved or were solved partially. These are outlined in this section such that they could be solved in future endeavours.

- The GRAPES algorithm combines temporal locality and popularity. But self-similar spatial locality is also a very important property of web request streams. Some self-similarity was observed in the GRAPES request stream. But values sometimes tended to differ between the R/S plots and variance time plot. This points at the non-stationarity stream. Most probably the approach followed by Huang and Devetsikiotis can solve the problem. They have proposed a method to combine short range and long range distributions.

- Studies need to be conducted on the efficiency of caching schemes. The experiments just concenterated on perfect LFU. The caching scheme used by caches are LRU and imperfect LFU. They need to be studied most probably using transforms. Then only caching performance can be accurately characterized and its relationships with latency need to be studied.

- We worked on a fuzzy logic algorithm that couldn't be completed. It basically revolved around the idea that there are certain caching servers that have other servers under their control. Such servers that act as leaders can trade other caching servers between them. Now when a request comes to any caching server it first multicasts it to members of its group and if it does not succeed then only it sends the request to its parent. Thus, caches are managed in groups and grouops can dynamically reconfigure. In this framework we tried to incorporate resource reservation and trading of some amount of cache of a server with another. Fuzzy logic was deemed most appropriate for managing such sort of a situation. The fuzzy variables were bandwidth, cache size, latency, average hit ratio and cache contents. The cache contents are compressed as a digest using bloom filters. To simulate bloom filters we had an idea that the rand function can be used as a URL hashing function with the page number as the seed. This algorithm should be developed and tested on a simulator.

- The proof of self-similarity([18, 19, 21]) in the web is very interesting. Please refer to Willinger and Taqqu's paper ([20]) that proves that superimposition of the traffic generated by individual users leads to overall self-similar traffic. The proof uses properties of Fractional Gaussian Noise and Fractional Brownian Motion. Theoretical studies on such functions and how they add up to generate self-similar traffic is a novel research direction.

References

[1] V. Almeida, A. Bestavros, M. Crovella, and A. D. Oliveira. Characterizing reference locality in the WWW. In Proceedings of 1996 International Conference on Parallel and Distributed Information Systems (PDIS '96), pages 92–103, December 1996.

[2] M.F. Arlitt and C.L. Williamson. Web server workload characterization: The search for invariants. In Proceeding of the ACM SIGMETRICS '96 Conference, Philadelphia, PA, April 1996.

[3] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In Proceedings of ACM SIGMETRICS Conference, July 1998.

[4] M.A. Blaze. Caching in Largescale Distributed File Systems. PhD thesis, Princeton Univ., Dept. of Computer Science, Jan. 1993.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips and S. Shenker. Web Caching and Zipf like Distributions: Evidence and Implications. IEEE Infocom '99.

[6] M.Busari and C. Williamson. On the Sensitivity of Web Proxy Cache Performance to Workload Characteristics. Proceedings of the IEEE Infocom Conference, New York, NY, pp. 126-134, March 1999.

[7] R. Carter and M. Crovella. Dynamic server selection using bandwidth probing in widearea networks. Technical Report BUCS96007, Boston Univ., Computer Science Dept., www.cs.bu.edu/techreports, Mar. 1996.

[8] A. Chankhunthod, P.B. Danszig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell. A hierarchical Internet object cache. In Proc. USENIX Annual Technical Conference, San Diego, California, Jan. 1996.

[9] A. Gibbons and W. Rytter. Efficient Parallel Algorithms. Cambridge University Press 1988.

[10] A. Heddaya, S. Mirdad and D. Yates. Diffusion-based Caching Along Routing Paths. Proc. 2nd Web Cache Workshop, Boulder, Colorado, June 9-10, 1997.

[11] Hypertext transfer protocol overview, http://www.w3.org/Protocols

[12] J. Jaja. Introduction to Parallel Algorithms. Addison Wesley 1992.

[13] The Network Simulator - ns-2, http://www.isi.edu/nsnam/ns/

[14] REAL 5.0 Overview, http://www.cs.cornell.edu/skeshav/real/overview.html

[15] SPECweb99 Benchmark, http://www.specbench.org/osg/web99/

[16] WebBench, http://www.etestinglabs.com/benchmarks/webbench/webbench.asp

[17] Why do research in Web Caching, http://www.web-caching.com/why-research.html

[18] J. Beran. Statistics for Long-Memory Processes. Chapman and Hall, New York,1994.

[19] M. Crovella and A. Bestavros. Selfsimilarity in world wide web traffic: Evidence and possible causes. In Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, May 1996.

[20] M. S. Taqqu, W. Willinger, and R. Sherman. Proof of a Fundamental Result in Self-Similar Traffic Modeling. Computer Communications Review 27(1997) 5-23

[21] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the Self-Similar Nature of Ethernet Traffic. ACM/SIGCOMM Computer Communications Review, Vol. 23, pp. 183–193, 1993. Proceedings of the ACM/SIGCOMM'93, San Francisco, September 1993.