The Measurement of Bandwidth: A Simulation Study

Martin J. Tunnicliffe
Kingston University
UK

1. Introduction

Bandwidth dictates the potential speed at which a channel carries information and hence the services it can support. Speech telephony for example requires only 4kHz bandwidth, while high quality sound requires 15kHz and video 6MHz (Glover & Grant, 2004). When several services share a common channel it is crucial to know the accessible bandwidth such that it can be divided efficiently between customers. Furthermore senders may even have to modify their behaviour in order to make best use of their bandwidth share (Yu et al., 2003). In an IP network bandwidths can be measured directly using the SNMP protocol to interrogate link components for their capacities and traffic loads: However since administrative access is not typically available to customers, numerous "end-to-end" measurement techniques have appeared whereby network traffic and infrastructure can be inferred from "probe" transmissions (Prasad et al., 2003). However, these techniques are only as good as their underlying assumptions and a plethora of misunderstandings and ambiguities have arisen (Jain & Dovrolis, 2004). Some of these problems are technological; wireless components introduce complications not experienced in wired networks (Johnsson et al., 2005) and traffic-shaping mechanisms can make the instantaneous bandwidth different from that experienced by sustained transmissions (Lakshminarayanan et al., 2004). However difficulties still arise under the simplest technological assumptions due to the complex behaviour of packet-streams within bottleneck links.

This chapter develops a simulation framework geared specifically towards bandwidth quantisation and measurement. The simulator does not represent any specific technology, but supports the networking of generic FIFO nodes on which the major probing philosophies (together with their various strengths and pitfalls) can be demonstrated. The reader is encouraged to experiment with the C++ source-code which is available online (http://staffnet.kingston.ac.uk/~ku12881/probesim1.0/).

The chapter is not fully comprehensive and several tools and techniques have not been covered. For more in-depth information the reader is directed to the references at the end of the chapter.

2. The Definition of Bandwidth

The exact meaning of "bandwidth" depends on the context in which it is used: In the analogue world it specifies the frequency range (in Hz) over which channel gain is principally effective; typically the difference between the upper and lower half-power points (though the definition used for noise calculations is slightly different). Bandwidth also has a separate digital definition: The maximum rate at which a channel can process data in bits per second.

Analogue and digital bandwidths are linked by the Shannon-Hartley equation which expresses the maximum error-free transmission rate R_{max} (bits/s) in terms of the analogue bandwidth B (Hz) and the signal-to-noise ratio S/N (simple ratio, not decibels):

$$R_{max} = B \cdot \log_2(1 + S/N) \text{ bits per second.} \quad (1)$$

R_{max} is directly proportional to B but falls to zero as $S/N \to 0$, indicating that a noisy channel has smaller capacity than a noiseless channel of the same bandwidth. By boosting the signal relative to the noise R_{max} can be increased indefinitely, though the logarithmic function introduces a law of diminishing returns. R_{max} represents the maximum *potential* digital bandwidth, though of course binary-coded data cannot necessarily be carried at this rate. In practical terms digital bandwidth may refer to a number of throughput-related concepts:

- *Link Capacity*. The maximum rate at which a specific link can operate.
- *End-To-End Bandwidth*. The bandwidth effective across a series of links.
- *Available Bandwidth*. The unused portion of link capacity accessible to new connections.

Figure 1 shows a hypothetical network path connecting source-node A to sink-node B. The path consists of four "hops" and passes through intermediate nodes C, D and E. Each link has a capacity of 100Mbit/s with the exception of DE which has only 50Mbit/s; this is the "narrow-link" which dictates the end-to-end capacity of the path. However, two cross-traffic flows are also present: A 70Mbit/s flow in the link CD and a 10Mbit/s flow in the DE. Thus the *available* bandwidths of CD and DE are 100-70=30Mbit/s and 50-10=40Mbit/s respectively, making CD the "tight-link" (the smallest available bandwidth) which dictates the end-to-end available bandwidth of the path. Note that the tight- and narrow-links are not necessarily the same.

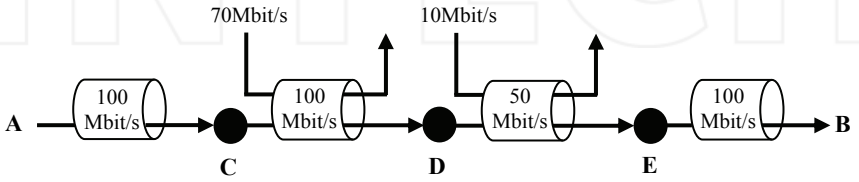


Fig. 1. A hypothetical 4-hop network path.

Often the best way to investigate a potential measurement algorithm is by computer simulation: The measurement can be compared directly with the "truth" (specified in the simulation parameters) and the investigator's "omniscience" provides information not otherwise accessible. It is easy for example to see the relative benefits of pairs and streams of probe-packets, the limitations of techniques based on the server idle-rate, the effects of multiple hidden infrastructural nodes and cost of bandwidth measurement in terms of increased utilisation and reduced quality-of-service.

3. The Simulation Software

All the simulations reported in this chapter were performed using a software system devised by the author. Readers are free to download this software, repeat the experiments using different parameters and modify the source code for their own purposes. The tool is not a program *per se* but a collection of classes which allow the user to write bespoke simulations in C++. Results are generated as text files which users can analyse using their preferred spreadsheet software. All the `.h` and `.cpp` files must be downloaded from http://staffnet.kingston.ac.uk/~ku12881/probesim1.0/ and added to the project, and the files `node.h`, `network.h`, `traffic.h`, `user.h`, `probe.h`, `connection.h`, `channel.h`, `simulation.h` and `constants.h` must be linked to any user-written code. There are eight classes in all: `network`, `node`, `channel`, `connection`, `user`, `probe`, `simulation` and `packet`. (The `packet` class is only ever used internally and will not be discussed here.) Since pseudorandom numbers are used, random seeding is required to ensure statistically independent repetitions. (Use `srand((unsigned)time(NULL))`.)

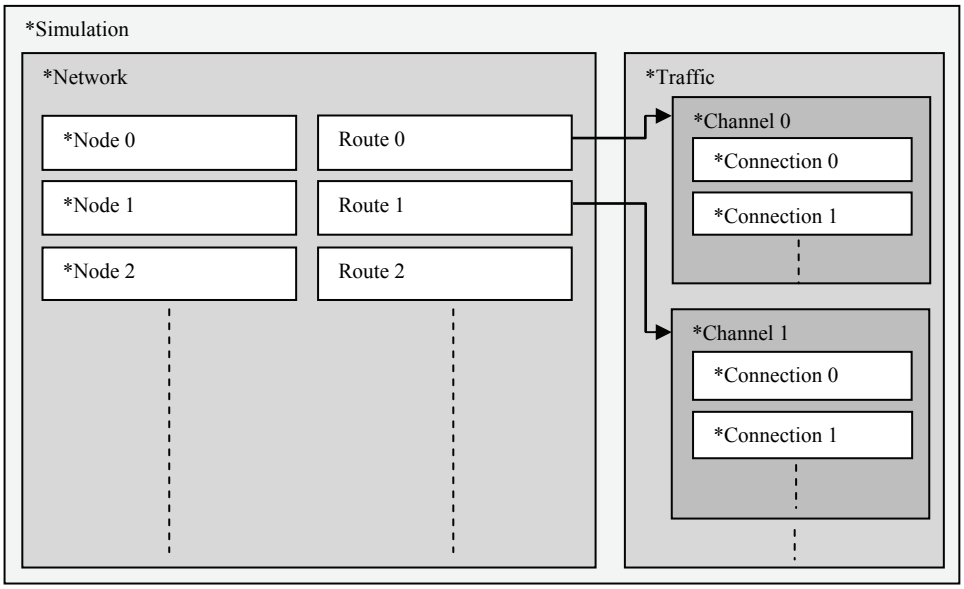


Fig. 2. Simulation architecture

3.1 The Simulation Class

Figure 2 shows the arrangements of objects within the `simulation` class, all of which are referenced by pointers. The class has two main attributes: A `network` object defining the network infrastructure and a `traffic` object defining the network traffic. It also has three public methods:

`simulation()` creates an empty unconfigured simulation object. The new object can perform no simulation function until its `configure` method has been called and returned `true`.

`bool configure(network *n, traffic *t)` gives the simulation network infrastructure `*n` and traffic `*t` and sets the virtual clock to zero. It returns `true` only if `*n` and `*t` are consistent with each other. Once it has been successfully called, the `advance` method (see below) may be used to make the simulation run.

`double advance(double duration)` advances the simulation's clock for a period of `duration` seconds, during which the behaviour of the network is simulated. Since a whole number of events must always be processed, the simulation time may overshoot the specified value, in which case the "actual" new time is returned. If the simulation fails (because it is not correctly configured) the returned time is identical to the time at which the method was called.

3.2 The Network Class

An object of the `network` class consists of nodes and routes. Nodes are objects of the `node` class (see below), and are identified by integers 0,1,2… which indicate the order in which they were added to the network. Routes (which represent allowed paths through the network) are strings of nodes' integer-identifiers. For example, `{0,1,2,-1}` passes through nodes 0, 1 and 2 before arriving at the sink node -1 (see Figure 3). The class has the following public methods:

`network()` creates an empty network with no nodes and no routes.

`bool insert_node(node *n)` adds a new node (identified by its pointer) to the network. It returns false if too many nodes are added (limit is set at 200 in `constants.h`) or if the method is parameterised with a NULL pointer.

`bool insert_route(int route[])` adds a new route to the network. The method returns false if too many routes are added (limit is set at 200 in `constants.h`), if any nonexistent or repeated nodes appear in the route, or if the array is not terminated by a -1.

`int get_no_of_nodes()` and **`int get_no_of_routes()`** return the number of nodes and routes added to the network.

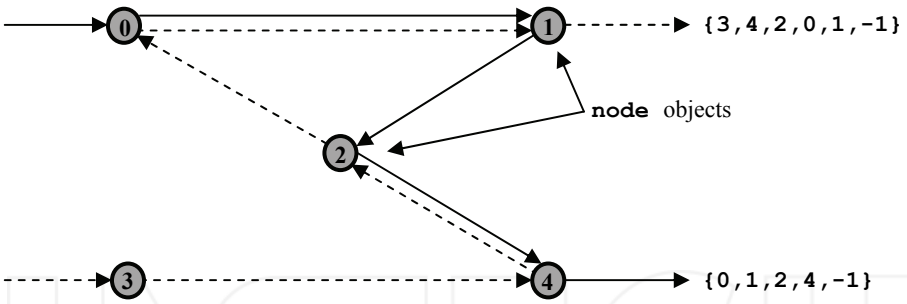


Fig. 3. Example topology specified by a `network` object.

3.3 The Node Class

An object of the `node` class is basically a FIFO (first in first out) queuing system which mimics the behaviour of a drop-tail router or store-and-forward switch. Its methods are as follows:

`node(int maxsize,double setspeed,double fixlatency)` creates a node which forwards data at `setspeed` bits/s, and adds an additional fixed delay `fixlatency` seconds to each packet. `maxsize` specifies the maximum amount of data (in bytes) that the node can hold before packets are dropped.

`double getidletime()` returns the time (in seconds) that the server has been idle since the start of the simulation, or the last time `resetidletime()` was called. This is useful for determining the server's average utilisation and idle-rate.

`void resetidletime()` sets the idle time to zero. This is useful for detecting changes in utilisation over time.

3.4 The Traffic Class

An object of the traffic class specifies the traffic applied to the network during the simulation. It consists of a collection of `channel` objects identified by integers 0,1,2…, indicating the order in which they were added. When loaded into a `simulation` object along with a network object, the channels in the traffic object correspond to the routes in the network object (channel 0 follows route 0, channel 1 follows route 1 etc, as shown in Fig.2). The class has the following public methods:

`traffic()` creates an empty traffic object with no channels.

`bool insert_channel(channel *new_channel)` inserts a channel object into the traffic.

3.5 The Channel Class

Channels specify the traffic applied to each route within the network: They are container-objects for connections, which specify actual traffic processes or transmissions. The channel class has two public methods:

`channel()` creates a new empty channel with no connections.

`bool insert_connection(connection *c`) places a new connection object within the channel.

3.6 The Connection Class

Objects of this class contain one of two attributes: A `user` object which specifies a user-related traffic process, or a `probe` object which measures the bandwidth of the path. It has three public methods of interest:

`connection()` creates a new empty connection object, containing neither a `user` nor a `probe`.

`bool insert_user(user *u)` and **`bool insert_probe(probe *p)`** insert a user or probe object into the connection, thereby deleting any existing user or probe object.

3.7 The User Class

An object of the `user` class represents an application which generates packets. Every `user` object is associated with a `connection` object (see above). In its default mode it generates a single packet of specified size, a specified number of seconds after the start of the simulation. However it can also be configured to generate a continuous stream of packets. Its public methods are as follows:

`user(int size,double start,bool trace)` creates a new `user` process which transmits one packet of `size` bytes, `start` seconds into the simulation. If the `trace` flag is set `true`, the progress of the packet through the network is tracked by a series of messages to the console window.

`bool set_stream(double r,bool p,char *d,char *t ,double startrec)` tells the user process to produce a continuous stream of packets at a rate `r` bits/s. If the flag `p` is set true then the stream will be governed by a Poisson (random memoryless) process; otherwise the packet interarrival times will be constant. The parameters `*d` and `*t` specify the filename/paths under which the packet delay and transmission times are to be stored. (If this data is *not* to be recorded, these pointers are set to `NULL`.) The parameter `startrec` specifies the time (from the beginning of the simulation) after which this recording is to commence.

`bool set_limits(double t1,double t2)` tells the user to generate packets only during the interval (`t1`, `t2`) seconds.

`bool pareto_modulation(double mean_on,double min_on,double mean_off)` switches the packet arrival mechanism to a Pareto ON-OFF process (Pitts & Schormans, 2000). the ON periods follow a Pareto distribution while the OFF periods are exponentially

distributed. The parameters specify the mean and minimum ON times as well as the mean OFF time.

`double get_packets_sent()`, **`double get_packets_carried()`** and **`double get_packets_dropped()`** are the accessor methods for the number of packets transmitted, carried and lost since the simulation began.

`double get_latency()`, **`double get_latency_stdv()`** and **`double get_jitter()`** are the accessor methods for the mean packet latency (delay), latency standard deviation and jitter in seconds. (Jitter is defined as the absolute difference between successive packet latencies.)

`bool closefiles()` closes the files whose names are specified in `set_stream`.

3.8 The Probe Class

Probe objects are used to measure the bandwidths of the connections into which they are inserted. The class has a number of methods, but these will be described later in conjunction with the bandwidth measurement algorithms to which they apply.

3.9 Some Typical Results

Figure 4 shows a three-hop network scenario, with link bandwidths of 5, 1 and 2 Mbit/s respectively. The 1Mbit/s link is clearly the narrow-link, though the 2Mbit/s is the tight-link due to its 1.5Mbit/s cross-traffic stream. Figure 5 shows the program listing which generates this infrastructure as a `network` object: Each network node has a maximum capacity of 100,000 bytes, and introduces zero additional delay per packet. The end-to-end route AB is represented by the route `rt0`.)

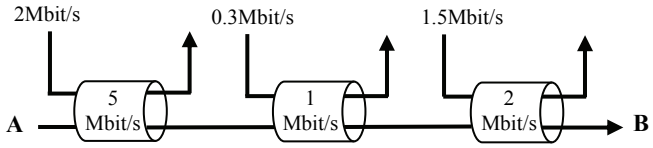


Fig. 4. Three hop network path

```
network *net=new network();

net->insert_node(new node(100000,5e6,0));
net->insert_node(new node(100000,1e6,0));
net->insert_node(new node(100000,2e6,0));

int rt0[]={0,1,2,-1},rt1[]={0,-1},rt2[]={1,-1},rt3[]={2,-1};

net->insert_route(rt0);
net->insert_route(rt1);
net->insert_route(rt2);
net->insert_route(rt3);
```

Fig. 5. Code to establish network infrastructure (routes and nodes).

```
//Set cross-traffic granularity
int Sc[]={60,148,500,1500};
double alpha[]={0.0477,0.0218,0.0950,0.8292};

//Establish cross-traffic processes
user *crosstraffic1[4],*crosstraffic2[4],*crosstraffic3[4];
for (int i=0;i<4;i++) {
   crosstraffic1[i]=new user(Sc[i],0,false);
   crosstraffic1[i]->set_stream(alpha[i]*2e6,true,NULL,NULL,0);
   crosstraffic2[i]=new user(Sc[i],0,false);
   crosstraffic2[i]->set_stream(alpha[i]*0.3e6,true,NULL,NULL,0);
   crosstraffic3[i]=new user(Sc[i],0,false);
   crosstraffic3[i]->set_stream(alpha[i]*1.5e6,true,NULL,NULL,0);
}

//Establish monitored process
user *thisuser=new user(500,0,false);
thisuser->set_stream(1e3,false,"d:\\delay.txt","d:\\time.txt",0);
```

Fig. 6. Code to establish traffic.

Fig.6 shows the code for generating the traffic processes. The process under observation is `thisuser` which sends 500 byte packets at 100kbit/s at equally spaced intervals (the `p` flag in `set_stream` is set to false) and record the packet launch-times and latencies to two `.txt` files on the d: root directory. The cross traffic is composed of four different packet-sizes: 60, 148, 500 and 1500 bytes, which constitute 4.77, 2.81, 9.5 and 82.92% of the total traffic respectively. (This profile was used by (Johnsson et al., 2005) and based on real observations.) Note that traffic of each packet size is generated by a separate Poisson `user` process. Figures 7 and 8 show the code to create the `traffic` object, and configure and run the simulation for 100 seconds.

```
//Establish cross-traffic connections
connection *crossconnect1[4],*crossconnect2[4],*crossconnect3[4];
for (i=0;i<4;i++){
   crossconnect1[i]=new connection();
   crossconnect2[i]=new connection();
   crossconnect3[i]=new connection();
}

//Associate cross-traffic processes with connections
for (i=0;i<4;i++){
   crossconnect1[i]->insert_user(crosstraffic1[i]);
   crossconnect2[i]->insert_user(crosstraffic2[i]);
   crossconnect3[i]->insert_user(crosstraffic3[i]);
}

channel *ch[4]; for (i=0;i<4;i++) ch[i]=new channel();
for (i=0;i<4;i++){
   ch[1]->insert_connection(crossconnect1[i]);
   ch[2]->insert_connection(crossconnect2[i]);
   ch[3]->insert_connection(crossconnect3[i]);
}

//Associate monitored process channel 0
connection *thisconnect=new connection();
thisconnect->insert_user(thisuser);
ch[0]->insert_connection(thisconnect);

//Establish traffic object
traffic *traff=new traffic();
for (i=0;i<4;i++) traff->insert_channel(ch[i]);
```

Fig. 7. Code to create connections and channels and create `traffic` object.

```
//Establish simulation
simulation *sim=new simulation();
sim->configure(net,traff);

//Run simulation
double endtime=sim->advance(100);
thisuser->closefiles();
```

Fig. 8. Code to create and run the simulation.

Figure 9 shows the latencies of `thisuser` packets, both as time-profiles and histogram distributions. (The `histogram.h` utility is available for producing these distributions.) Notice that the performance deteriorates as the transmission rate approaches 500kbit/s, the end-to-end available bandwidth of the path. Close to and exceeding this rate the buffer begins to overflow and packets are dropped at the tight-link buffer. This illustrates that the tight-link is the most important bottleneck under any specific loading conditions, though the narrow-link capacity represents the "best case" scenario when cross traffic is at a minimum.

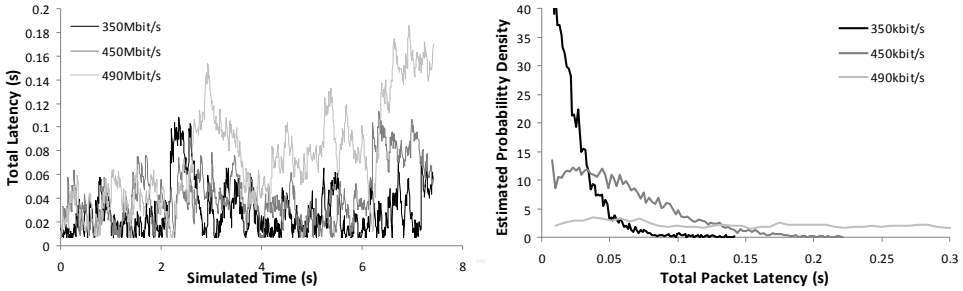


Fig. 9. Simulated packet delay profile and distribution for the three-hop network of Fig. 4.

4. Taxonomy of Bandwidth Measurement Algorithms

It will be recalled that bandwidth estimates may refer to the total link capacities or available bandwidth of individual hops or of end-to-end paths. There are many specific *tools* available for bandwidth measurement such as *pathload* (Jain et al. 2002) and *pathchirp* (Ribeiro et al., 2003) these are based on a limited number of fundamental approaches. Here we examine four such approaches: Idle-Rate, Packet Pair/Train Dispersion (PPTD), Self-Loading Periodic Streams (SLoPS) and Trains of Packet Pairs (TOPP). (A fifth approach Variable Packet Size (VPS) probing is also worth discussing but its reliance on IP technology puts it somewhat outside the scope of this chapter.) The approaches may be classified according to the quantities they aim to measure (see Table 1).

	Link Capacity	**Available Bandwidth**
Per Link	Iterative TOPP/VPS	Iterative TOPP, Idle Rate
End-To-End	PPTD	SloPS/TOPP

Table 1. Classification of Bandwidth Measurement Algorithms

5. Narrow-Link Measurement: PPTD

The Packet Pair/Train Dispersion (PPTD) technique measures the end-to-end capacity of a path using the "bottleneck spacing effect". It has been studied for some years; one of the earliest and most thorough investigations was published as early as ten years ago (Lai & Baker, 1999). Multiple pairs (or trains) of probe packets are sent back-to-back along the monitored route and their dispersion (the time difference between the last bit of each packet) is measured. First consider a single network link *without* cross-traffic: If the input and output dispersions are Δ_{in} and Δ_{out} seconds then

$$\Delta_{out} = \max\left(\Delta_{in}, L/C\right) \text{ seconds} \tag{2}$$

where L is the packet size in bits and C is the link capacity (bits/s). We shall refer to the condition $\Delta_{out} = \Delta_{in}$ as "sub-congestion" and $\Delta_{out} = L/C$ "congestion"; it is under the latter

that the link capacity can be calculated as L/Δ_{out} bits per second. Figure 10 illustrates this for the simple case of a packet pair.

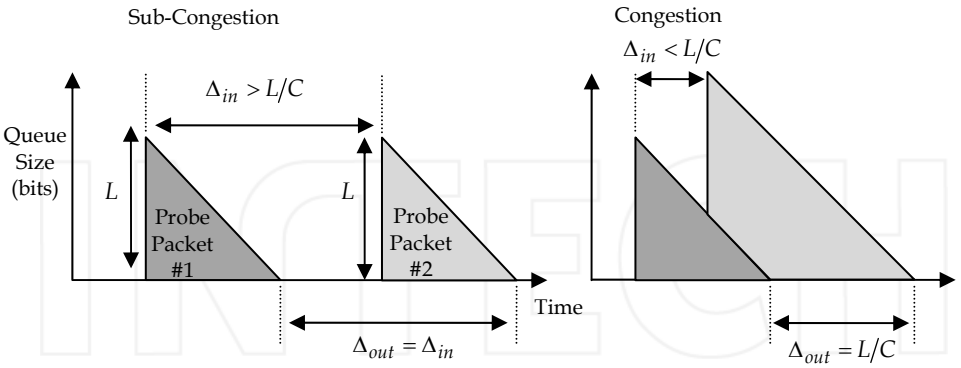


Fig. 10. Packet pair without cross-traffic.

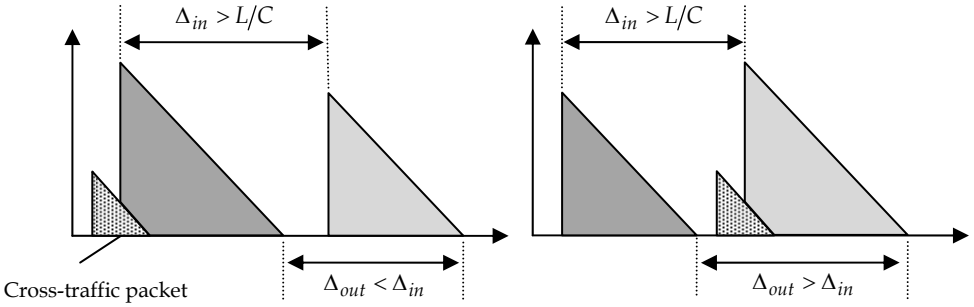


Fig. 11. Cross-traffic in a sub-congested link: Output dispersion may be pushed below or above Δ_{in}.

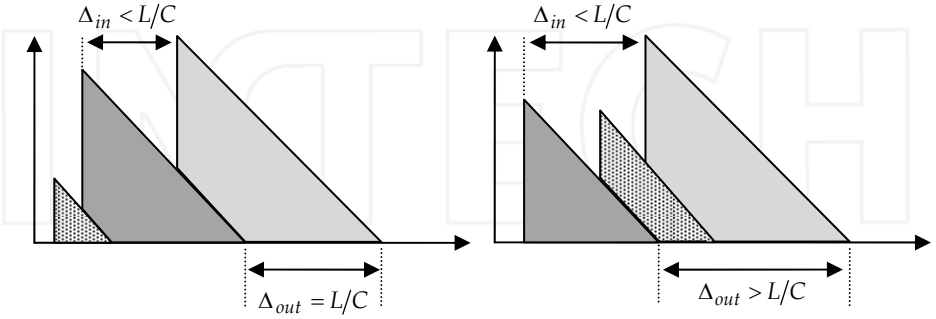


Fig. 12. Cross-traffic in a congested link: Dispersion may only be forced *above* L/C.

Figures 11 and 12 illustrate (with some simplification) how cross-traffic interferes with dispersion: In sub-congestion the output dispersion can be increased ("dialated") or decreased ("compressed") relative to the value given by Eqn.2, depending on which of the two packets is delayed the more. However, under congestion the output dispersion can only be dialated by the presence of cross-traffic. Now let C_{min} be the capacity of the narrow link in the path: As Δ_{in} is decreased gradually from a large value, the effective probing rate $R = L/\Delta_{in}$ increases until it exceeds C_{min} at which point it pushes the narrow-link into congestion. Now since the nodes upstream of the narrow-link are sub-congested, R may be above or below L/Δ_{in} when the probe-pair reaches the narrow-link. However, the node immediately downstream of the narrow-link experiences a *minimum* input dispersion of L/C_{min} which (as it is also sub-congested) it may cause to increase or decrease. The overall process is shown in Figure 13.

To simulate a PPTD, a `probe` object is created using the following methods of the probe class:

`probe(double separation,int measurements,bool trace)` creates a new probe. The parameters `measurements` and `separation` specify the number of probing measurements and the time separation between them. If the `trace` flag is set true, the progress of each probe packet is traced through the network by a series of messages to the console window.

`bool set_PPTD(int L,double din,char *dout,char *b,char *d1,char *d2)` configures the probe for PPTD measurement using pairs of probe packets `L` bytes long with an input dispersion `din` seconds. The remaining parameters specify filenames for recording the output dispersion Δ_{out} (`dout`), the bandwidth estimation L/Δ_{out} (`b`) and the end-to-end latencies for the first (`d1`) and second (`d2`) packet of each packet pair.

`bool set_streamsize(int stream_size)` tells the probe to use streams of `stream_size` packets. (The default stream-size is 2.)

Figure 14 shows the code modification (relative to Figures 7 and 8) required to probe the three-node network path with 1,000 pairs of spaced 0.1 seconds apart. (The methods **`openfiles()`** and **`closefiles()`** merely open and close the files **`odisp.txt`**, **`bw.txt`**, **`del1.txt`** and **`del2.txt`** where the probe-packet data are stored.)

Figure 15 shows the distributions of bandwidth estimations for three different packet sizes for a probing rate $> C_{min}$ (R was held constant at 1.6Mbit/s). C_{min} =1Mbit/s appears as a local maximum within each distribution: The maxima below C_{min} may be caused by dispersion "dialation" at nodes upstream or downstream of the narrow-link, but the modes *above* C_{min} must be caused by dispersion "compression" in downstream nodes. The "true" bandwidth can be identified since it maintains its position as the packet size is changed (Pasztor & Veitch, 2002). Notice that when the packet size is small, the distribution modes (both true and spurious) are much crisper and better defined than when the packet size becomes larger, and the true result ceases even to be the dominant mode of the distribution.

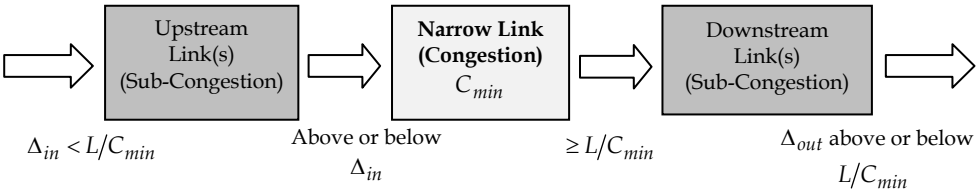


Fig. 13. Variation of packet dispersion across path where only the narrow-link is in congestion.

```
probe *thisprobe=new probe(0.1,1000,false);
thisprobe->set_PPTD(100,0.5e-4,"d:\\odisp.txt","d:\\bw.txt",
                                "d:\\del1.txt","d:\\del2.txt");
connection *thisconnect=new connection();
thisconnect->insert_probe(thisprobe);
ch[0]->insert_connection(thisconnect);
traff->insert_channel(ch[0]);
simulation *sim=new simulation();
sim->configure(net,traff);

thisprobe->openfiles();
double endtime=sim->advance(100);
thisprobe->closefiles();
```

Fig. 14. Code to to create and run PPTD probing experiment.

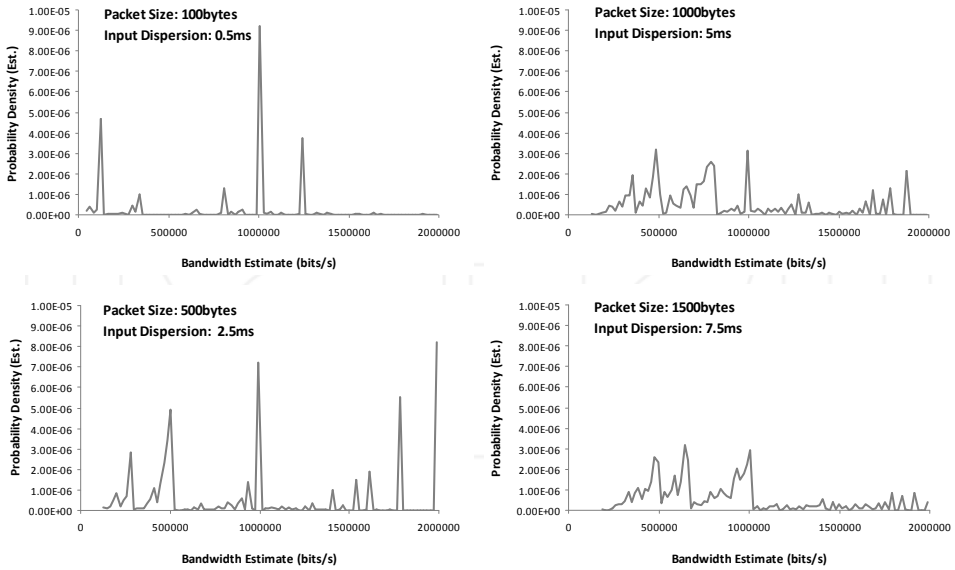


Fig. 15. Estimated bandwidth distributions for three-node path (Fig. 4) using 1.6Mbit/s probing rate.

6. Tight-Link Measurement: SLoPS

Like PPSD, the Self Loading Packet Stream (SLoPS) technique uses probe-packet streams, but measures end-to-end available bandwidth A_{min} instead of link capacity (Jain & Dovrolis, 2003). If a short-lived stream of equal-sized packets is sent at R bits/s, variations in one-way delays give an indication of congestion buildup. When R is greater than A_{min} short-term overload causes the one-way delay to increase steadily with time. If $R < A_{min}$ this is not the case and (once equilibrium is reached) the average delay remains approximately constant. The sender usually uses a binary search algorithm to probe the path at different rates, while the receiver reports the resulting delay-trends. Though the `probe` class has (as yet) no dedicated methods devoted to SloPS, individual packet delay files produced by the `user` object may be used to investigate certain features of SloPS.

Figure 16 shows the code used to generate streams of 63 packets over a mid-simulation interval of 0.1 seconds and measure the end-to-end latency of each packet. Figure 17 shows the delay trends (averaged for 5 independent simulations) for three different probing rates. An unambiguously increasing trend is only seen when $R > A_{min}$.

One drawback of SloPS is that it requires accurate end-to-end delay measurement, which in turn demands the accurate synchronised end-point clocks. PPTD does not suffer from this difficulty as it uses relative inter-packet times rather than absolute end-to-end times.

```
user *thisuser=new user(100,0,false);
thisuser->set_stream(500e3,false,"d:\\delay.txt","d:\\time.txt",0);
thisuser->set_limits(50,50.1);

connection *thisconnect=new connection();
thisconnect->insert_user(thisuser);
ch[0]->insert_connection(thisconnect);
traff->insert_channel(ch[0]);

simulation *sim=new simulation();
sim->configure(net,traff);

double endtime=sim->advance(100);
thisuser->closefiles();
```

Fig. 16. Code for SloPS demonstration. (Packet size 100bytes, stream rate 500kbit/s.)

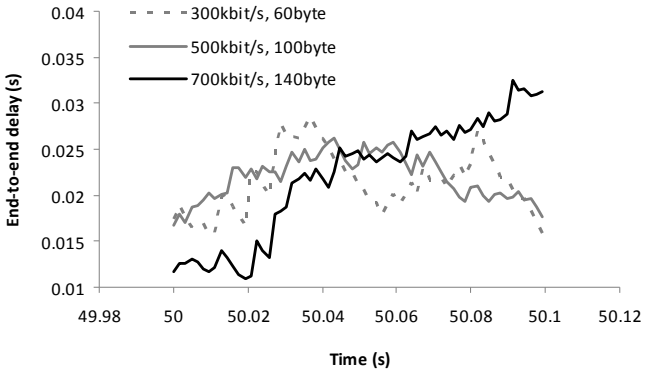


Fig. 17. Packet delay trends for three-node path (Fig. 4.) loaded with 300, 500 and 700kbit/s streams over a 100ms time-slice. Only the 700kbit/s stream shows a consistently increasing delay-trend. (Available bandwidth is 500kbit/s.)

7. Tight-Link Measurement: Idle Rate

This technique was proposed specifically for broadband access networks where the effects of self-induced packet congestion can yield misleading results (Lakshminarayanan et al., 2004). For a single link in isolation, the available bandwidth A is related to tight-link capacity C and server utilisation ρ by the formula

$$A = C \cdot (1 - \rho) \text{ bits per second.} \tag{3}$$

where $(1-\rho)$ is the "idle rate", i.e. the ratio of time during which the link is inactive. If C is known then A can be calculated simply by multiplying by the idle rate, which can (in principle) be inferred from the delays of periodically-transmitted probe-packets. If these are plotted as a cumulative frequency distribution then the idle-rate should be visible as a "knee" in the graph where the latency begins to increase.

This works quite well for isolated links: Figure 1(a) has link capacity of 1Mbit/s and an available bandwidth of 500kbit/s. The cumulative plot in Figure 18(c) shows a knee (idle rate) at 0.5 and A =0.5×1Mbit/s=500kbit/s. However, the technique is not always so successful: Figure 18(b) shows the same path with a 2Mbit/s upstream link added: The corresponding cumulative frequency curve suggests a tight-link idle-rate of 0.25, giving an available bandwidth estimate of 0.25×1Mbit/s =250kbit/s; half its true value.

Aside from the obvious disadvantage that it requires prior knowledge of link capacity, the idle-rate method is clearly not always usable when there are moderately utilised links other than the tight-link. Also like SloPS it relies upon a capacity for end-to-end delay measurement.

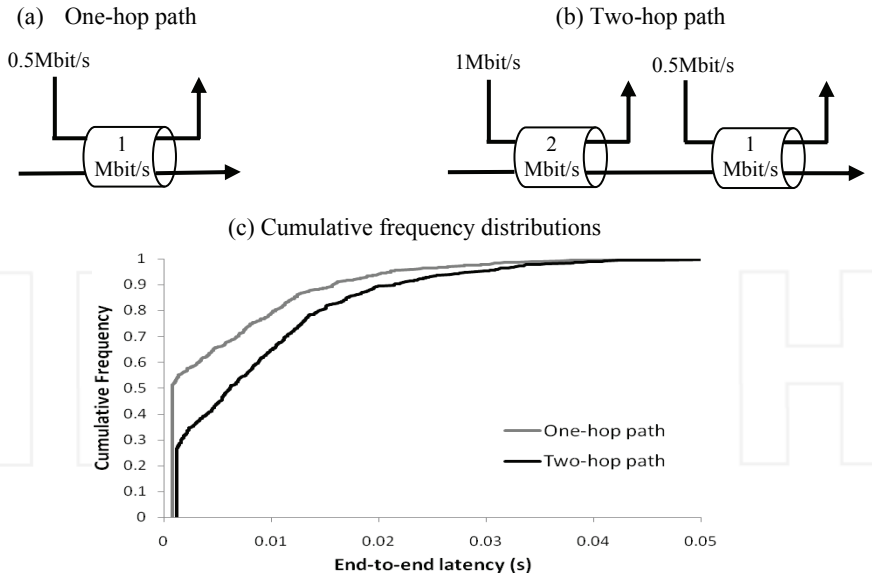


Fig. 18. (a) One node and (b) two node network paths each with 500kbit/s available bandwidth, (c) Cumulative frequency plots for end-to-end delays.

8. Tight-Link Measurement: TOPP

Like SloPS, the Train Of Packet Pairs algorithm (TOPP) measures end-to-end available bandwidth. Many packet pairs (or streams) are sent with a gradually decreasing dispersion: If Δ_{in} is the input dispersion and L the packet-size (bits) then the *offered rate* $R = (n-1)L/\Delta_{in}$. If this exceeds the end-to-end available bandwidth A then momentary congestion causes each packet to be be delayed (on average) longer than its predecessor. This increases the output dispersion Δ_{out} and decreases the *measured rate* $M = (n-1)L/\Delta_{out}$ (see Figure 19). On the other hand if $R < A$ then the average offered and measured rates should remain approximately equal ($R/M \approx 1$).

If only one bottleneck is visible then the governing equation can be shown to be (Melander et al., 2000).

$$\frac{R}{M} = \frac{\Delta_{out}}{\Delta_{in}} = \min\left(1, \frac{1}{C} \cdot R + \left[1 - \frac{A}{C}\right]\right). \tag{4}$$

Rearrangement of Eqn.(4) allows the link capacity and available bandwidth to be estimated from the slope and intercept of a regression lines fitted to the graph-segment $R > A$:

$$C = \frac{1}{SLOPE} \qquad A = \frac{1 - INTERCEPT}{SLOPE}$$

One difference between TOPP and SloPS is that the former increases the offered rate linearly while the latter uses a binary search algorithm. Another important difference is that TOPP can estimate the *capacity* of the tight link as well as the effective bandwidth.

To simulate TOPP, a further method of the `probe` class is called:

`bool set_TOPP(int L,double r1,double r2,double dr,char *r,char *m,char *v)` configures the probe for TOPP measurement using pairs of probe packets `L` bytes long with an input rate starting at `r1` bits/s and increasing to `r2` bits/s in intervals of `dr` bits/s. The remaining parameters specify filenames for recording the probing rate (`r`), the mean output dispersion ratio $\Delta_{out}/\Delta_{in} = R/M$ (`m`), and the variance of the latter (`v`).

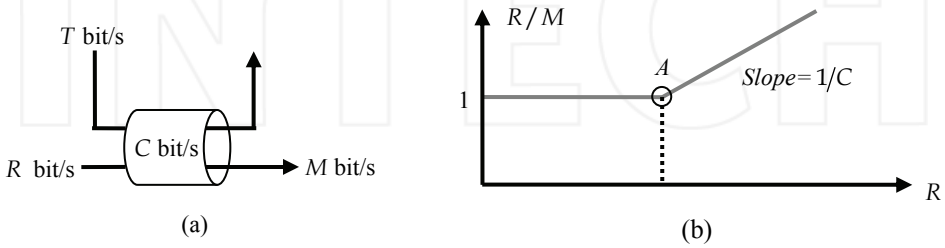


Fig. 19. The TOPP algorithm. (a) The available bandwidth is the link capacity C minus the cross traffic T. (b) A linearly increasing R/M with increasing R indicates that $R > A$.

Figure 20 shows the code for a simple TOPP experiment, and Figure 21 some results for a single hop path. The graphs roughly resemble Figure 19(b), though there is no abrupt transition between the two linear domains. This is due to the finite granularity of the cross-traffic (Park et al., 2006) which also affects the measured slopes of the graphs. Regression lines were used to obtain the estimates for C and A listed in Table 2, which show a tendency to overestimate C and underestimate A - particularly when the stream-size is small. This effect is is well documented and is commonly referred to as the "probing bias" (Liu et al., 2004).

```
probe *thisprobe=new probe(0.5,100,false);
thisprobe->set_TOPP(500,0.01e6,2e6,0.01e6,"d:\\rate.txt",
                    "d:\\mean.txt","d:\\var.txt");
thisprobe->set_streamsize(10);

connection *thisconnect=new connection();
thisconnect->insert_probe(thisprobe);
ch[0]->insert_connection(thisconnect);
traff->insert_channel(ch[0]);

simulation *sim=new simulation();
sim->configure(net,traff);

thisprobe->openfiles();
double endtime=sim->advance(50000);
thisprobe->closefiles();
```

Fig. 20. Code to to create and run TOPP probing experiment.

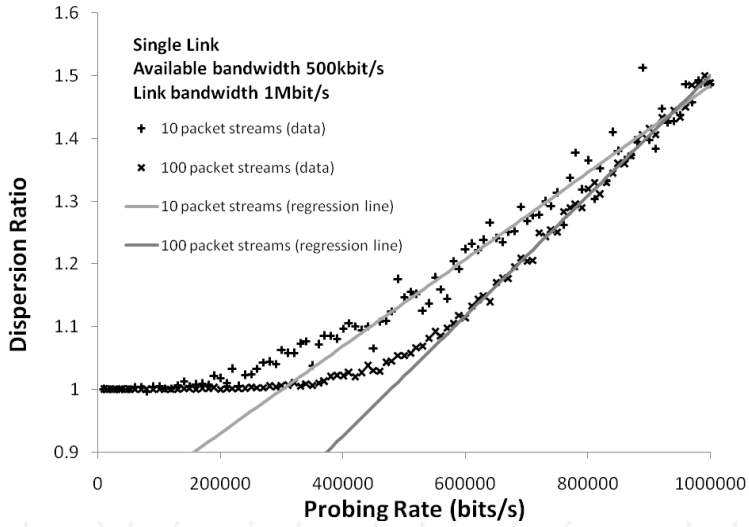


Fig. 21. Results TOPP results obtained by probing a single link with a total bandwidth of 1Mbit/s and an available bandwidth of 500kbit/s. Probe packets were 500 bytes each.

When applied to a multiple-hop network path, the graph displays multiple slope-changes (Figure 22). An iterative approach has been explored (Melander et al., 2002) whereby the available bandwidths of multiple bottlenecks can be inferred, using positive spikes in the second derivative $\partial^2(R/M)/\partial R^2$ to indicate the slope-changes. However this technique relies on prior assumptions about the order in which the bottlenecks appear and a policy of Shortest Surplus First (the smallest available bandwidth is assumed to be the furthest upstream) has been adopted as a worst-case scenario.

Stream Size (Packets)	Link Capacity (Mbit/s)	Available Bandwidth (Mbit/s)
10	1.440	0.300
100	1.040	0.477
True Value:	1.000	0.500

Table 2. TOPP bandwidth estimates for different sized streams of 500 byte packets.

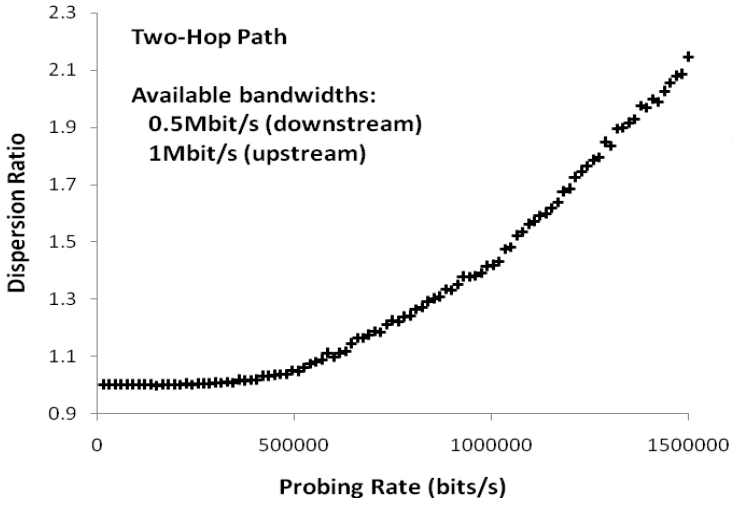


Fig. 22. Results TOPP results obtained by probing a double-hop network path with 100-byte packet streams. Slope changes at 0.5 and 1Mbit/s indicate the available bandwidths of the two bottlenecks.

9. Conclusions

This chapter presents a simulation framework for studying bandwidth quantisation and measurement. It does not represent any specific technology but rather an interconnection of generic FIFO queuing nodes which can approximate the behaviour of real networks. The reader is invited to experiment with the C++ source-code which is available online. The four algorithms investigated were PPTD, SLoPS, Idle Rate and TOPP. A fifth approach is Variable Packet Size (VPS) probing: This measures round-trip-times for a series of probe packets to each intermediate node (Prasad et al, 2003) using ICMP messages to signal back to the sender. By varying the packet size L, the serializdation delay L/C can be separated from the independent propagation-time and the effects of queuing delay minimised by taking the minimum of several measurements. Though this can provide quite detailed information, it assumes Layer 3 functionality in all congestible links. The technique has therefore not been included in this study.

The aim of this chapter is not to present any particularly significant new insights, nor to reproduce the full complexity of past research (which would require an entire book). We

aim rather to show how a relatively simple tool can be used to demonstrate the principles and provide a starting-point for the reader's own investigations.

10. References

Glover, I.A. & Grant,P.M. (2004), *Digital Communications* (2nd Ed.), Pearson Prentice Hall, p.3.

Jain, M.; Dovrolis, C. & Mah, B. (2002). Pathload: An Available Bandwidth Estimation Tool, *Proceedings of Passive and Active Measurement Conference*.

Jain, M. & Dovrolis, C. (2003). End-to-End Available Bandwidth Measurement Methodology, Dynamics and Relation with TCP Throughput, *IEEE/ACMA Transactions on Networking*, Vol. 11, No. 4, pp. 537-49.

Jain, M. & Dovrolis, C. (2004). Ten Fallacies and Pitfalls on End-to-End Available Bandwidth Estimation, *Proceedings of 4th. ACM SIGCOMM*, pp.272-7, Taormina, Sicily.

Johnsson, A.; Melander, B. & Björkman, M. (2005) Bandwidth Measurement in Wireless Networks, *Proceedings of Mediterranean Ad Hoc Networking Workshop*, Porquerolles, France, June 2005.

Lai, K. & Baker, M. (1999). Measuring Bandwidth, *Proceedings of IEEE INFOCOM 1999*, pp. 905-14.

Lakshminarayanan, K; Padmanabhan, V.N. & Padhye, J, Bandwidth Estimation in Broadband Access Networks, *Proceedings of 4th ACM SIGCOMM Internet Measurement Conference* (IMC'04), Taormina, Italy, October 2004, pp. 314-21.

Liu, X.; Ravindran, K.; Liu, B.; Loguinov, D. (2004). Single-Hop Probing Asymptotics in Available Bandwidth Estimation: Sample Path Analysis, *Proceedings of ACM Internet Measurement Conference 2004*.

Melander, B.; Björkman, M. & Gunningberg, P. (2000). A New End-to-End Probing and Analysis Method for Estimating Bandwidth Bottlenecks, *Proceedings of IEEE Glogecom'00*, San Francisco, CA, USA, Nov. 2000.

Melander, B.; Björkman, M. & Gunningberg, P. (2002). Regression-Based Available Bandwidth Measurements, *Proceedings of 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*.

Park, K.J.; Lim, H. & Choi, C-H. (2006). Stochastic Analysis of Packet-Pair Probing for Network Bandwidth Estimation, *Computer Networks*, Vol. 50, pp. 1901-15.

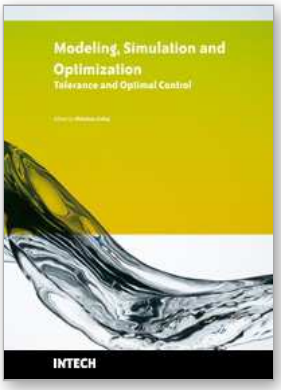
Pasztor, A. & Veitch, D. (2002b). The Packet Size Dependence of Packet Pair Like Methods, *Proceedings of IEEE/IFIP International Workshop on Quality of Service (IWQoS)*, 2002.

Pitts, J.M. & Schormans,J.A. (2000). IP and ATM Design and Performance, Wiley, pp. 287-98.

Prasad, R.S.; Murray, M; Dovrolis, C. & Claffy, K. (2003). Bandwidth Estimation: Metrics, Measurement Techniques and Tools, *IEEE Network*, Vol.17, No.6, pp.27-35.

Ribeiro, V.J.; Riedi, R.G.; Baraniuk, J.; Navratil, L. & Cottrel, L. (2003). Pathchirp: Efficient Available Bandwidth Estimation for Network Paths, *Proceedings of Passive and Active Measurement Workshop*, 2003.

Yu, Y.; Cheng, I. & Basu, A. (2003). Optimal Adaptive Bandwidth Monitoring for QoS Based Retrieval, IEEE Transactions on Multimedia, Vol. 5, No. 3, pp. 466-73.



Modeling Simulation and Optimization - Tolerance and Optimal Control

Edited by Shkelzen Cakaj

ISBN 978-953-307-056-8
Hard cover, 304 pages
Publisher InTech
Published online 01, April, 2010
Published in print edition April, 2010

Parametric representation of shapes, mechanical components modeling with 3D visualization techniques using object oriented programming, the well known golden ratio application on vertical and horizontal displacement investigations of the ground surface, spatial modeling and simulating of dynamic continuous fluid flow process, simulation model for waste-water treatment, an interaction of tilt and illumination conditions at flight simulation and errors in taxiing performance, plant layout optimal plot plan, atmospheric modeling for weather prediction, a stochastic search method that explores the solutions for hill climbing process, cellular automata simulations, thyristor switching characteristics simulation, and simulation framework toward bandwidth quantization and measurement, are all topics with appropriate results from different research backgrounds focused on tolerance analysis and optimal control provided in this book.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Martin J. Tunnicliffe (2010). The Measurement of Bandwidth: A Simulation Study, Modeling Simulation and Optimization - Tolerance and Optimal Control, Shkelzen Cakaj (Ed.), ISBN: 978-953-307-056-8, InTech, Available from: http://www.intechopen.com/books/modeling-simulation-and-optimization-tolerance-and-optimal-control/the-measurement-of-bandwidth-a-simulation-study

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821