Dummynet and Forward Error Correction

Luigi Rizzo
Dip. di Ingegneria dell'Informazione – Universita' di Pisa
`http://www.iet.unipi.it/~luigi/`
Email: l.rizzo@iet.unipi.it

Abstract

In this paper we present a couple of tools developed by the author on FreeBSD, and available from the author's Web page in source format. The first one, called dummynet, is a tool designed for the performance evaluation of network protocols and applications. Despite its original design goal, there has been a lot of interest on using dummynet as a bandwidth manager in network servers. dummynet simulates the effect of finite queues, bandwidth limitations, and queueing delays, and is embedded in the protocol stack of the host, allowing even complex experiments to be run on a single machine, using existing applications and protocol implementations.

The second tool is a software implementation of an erasure code especially suited for use in network protocols. Erasure codes are used in Forward Error Correction (FEC) techniques to reduce or remove the need for retransmissions in presence of communication errors. FEC has been rarely used in network protocols, because of the encoding/decoding overhead, and also because the underlying theory of error correcting codes is generally not well known to network researchers. In this paper we discuss the theory behind a simple erasure code, and provide performance data to show that the encoding/decoding overhead is acceptable for many applications even on low-end machines.

1 Introduction

With computer networks becoming more and more widespread, there is an increasing number of distributed applications designed to run across networks with the most different features. This variability of operating conditions poses severe challenges to designer of applications, especially in the testing phase, where the interaction of the application with networks and communication protocols must be studied to point out potential problems or unexpected behaviours.

The first tool presented in this paper, called dummynet, covers the need, that every designer has, to evaluate the behaviour of a protocol, or an application, in a real network environment. Simulation is often not an option, because of the requirement to build a simulation model of the system under test, whose features might not be fully known. Experiments on a real network might be problematic as well, because of the unavailability of a suitable testbed, or difficulties in configuring the testbed itself. dummynet solves many of these problems, by merging the advantages of simulation and testing on real networks, thus constituting a flexible and cheap testing tool.

Another common problem in the development of distributed applications is to recover from errors and lost packets. This task is generally accomplished by retransmitting missing packets on demand (this technique is called ARQ). There are situations where the use of ARQ is impractical: e.g. with mobile equipment, sending the retransmission requests to the base station drains precious power from the battery; on unidirectional channels such as broadcast satellite links, an uplink channel is simply not available; and in multicast communication, ARQ might not scale as well as we would like, because of the presence of uncorrelated losses at different receivers.

An alternative to ARQ relies on encoding data in a redundant way, such that the receiver can reconstruct the original data even in presence of missing packets. This technique, called FEC, is especially useful for reliable multicast protocols, and/or for highly asymmetric communication channels. However, FEC has been rarely used in networking protocols [1], because the encoding/decoding procedures are commonly believed to be very expensive, and possibly also because the principles of operation of such procedures are not well known to the the non-specialist in coding theory.

The encoder/decoder presented in this paper shows that FEC can be implemented in software with a reasonable performance on today's hardware, thus opening the way to the design of protocols based on FEC rather than pure ARQ. While we try to provide enough details to let the designer fully understand the operation of our encoder, it is also possible to use our implementation as a black box in building new applications.

The paper is structured as follows. In Section 2 we present dummynet, starting with a brief description of its principle of operation, followed by a discussion of its implementation and possible applications. Section 3 provides a description of our erasure code: the theory behind the operation of erasure codes is first presented in Sections 3.1 and 3.2, followed by a discussion of various implementation issues. Performance data of our implementation of the code are shown in Sec. 3.3. Finally, Section 3.4 discusses some applications of our code.

2 Dummynet

The study of implementations of network protocols and application is often done by simulation or by running experiments in a real network. Both approaches have their pros and cons. Simulations require the development of a simulation model of the system under analysis. The unavoidable inaccuracies in the model might adversely affect the results of experiments, and perhaps even prevent the detection of features of the actual implementation (including bugs). Experiments on a real network, on the other hand, require the availability of a suitable testbed, which might not exist or might be hard to configure correctly.

To tackle the problem, some simplifications can be made, e.g. considering all phenomena of interest (queueing, delays, bandwidth constraints) confined to one or a few bottleneck links, and trying to run experiments or simulations on such topologies. Packet level simulators have been built to this purpose[6, 4, 9], and successfully used in research. As an alternative, experiments on real systems are performed using modified routers acting as a "flakeway", and configured to introduce delay, losses and other perturbations to the traffic. The tool proposed in this paper acts much like a flakeway, except that it operates within the protocol stack of the system used for experiments, rather than in an external router.

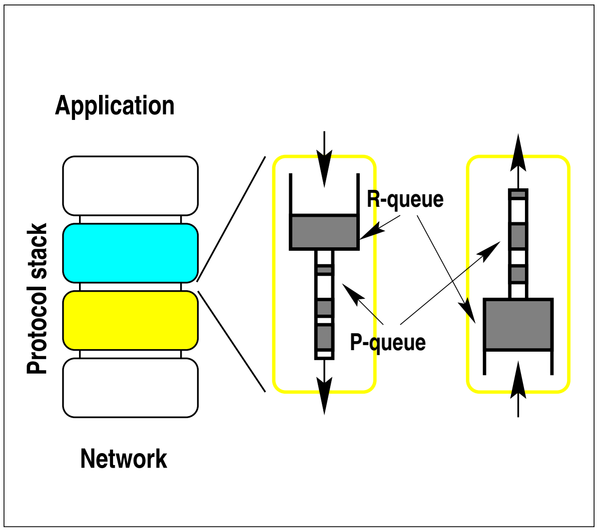

Figure 1: dummynet operates by intercepting traffic in the protocol stack of the host system, and simulating the effects of queueing, bandwidth limitations and propagation delays.

2.1 Principle of operation

The presence of a real network between communicating processes affects traffic by one or more of the following mechanisms:

- propagation delays;
- queueing delays, introduced by bandwidth-limited communication channels;
- losses, due to the queue overflows and (to a lesser extent, at least in wired networks) noise;
- packet reordering, due to the presence of multiple paths between the sender and the receiver.

Most if not all of these phenomena can be replicated by intercepting packets in their way in or out of a protocol stack, and simulating delays, losses and reordering.

In particular, the effects of bounded-size queues, propagation delays, and bandwidth-limited channels can be simulated by passing packets through a couple of queues, named R-queue and P-queue, (see Fig. 1). The queues are inserted at a convenient point in the protocol stack (in our implementation, this occurs at the upper interface of IP). They implement what we call a *pipe*, characterised by a bandwidth B, propagation delay t_P, and queue size S. The rules to move packets between queues are the following:

- Packets are first put into the R-queue, which simulates the effect of the bounded-size queues that are usually found in front of a network interface. Insertions in this queue will be done according to the queueing policy of choice (FIFO with tail-drop in our case, so at most S packets can be in the R-queue at any time).
- Packets are extracted from the R-queue and moved into the P-queue, at a maximum rate of B bytes per second. This simulates the bandwidth limitations of the communication link.
- Packets remain in the P-queue for t_P seconds, to simulate the propagation delay associated to the link. After the delay, packets are delivered to the next layer in the protocol stack.

Packet reordering can be conveniently simulated by modifying the policy for insertions in the R-queue. Losses due to noise or interference are similarly easy to introduce, by dropping packets at random in their flow through the queues.

The transitions of packets between queues are performed by a periodic task run every T second. At each run, the task moves at most BT bytes from the R- to the P-queue, and extracts packets from the P-queue after they have been there for t_P/T cycles. All queue operations require constant time, and this permits running the periodic task at a conveniently high frequency with limited overhead.

2.2 Implementation

dummynet, as presented in the previous Section, has been originally developed in FreeBSD in late 1996 as a module to intercept TCP traffic [14], and has been used by the author for research and didactical purposes. The original tool only included a pair of pipes, configured with the `sysctl` command.

The core of dummynet is made of about 400 lines of code, implementing the R- and P- queues and the procedures to move packets around. Each pipe is described by a record containing the parameters of the pipe, and pointers to the R- and P- queues. Elements of each queue are the packets to be passed to `ip_input()` and `ip_output()`, plus any additional argument required by these functions. The periodic task to move packets between queues is only active when there are packets in any pipe, and is run `HZ` times per second (we often ran with `HZ=1000` even on 486-class machines). We have written the code in such a way to reduce dependencies on other parts of the operating system, so that the code can be easily ported of other systems with little effort.

The simplicity and low overhead of dummynet soon suggested its use as a general purpose bandwidth manager/traffic shaper for network servers, and we have received several requests of this kind since we made our code publicly available. In order to transform dummynet into a practical bandwidth manager, a flexible configuration and packet filtering mechanism was needed. Rather than writing a special purpose module for packet filtering, we have used for this purpose the `ipfw` firewall code already present in FreeBSD. This gives us the advantage of reusing existing code and configuration methods that the user can be already familiar with, avoids the duplication of functionalities in the system, and allows us to benefit of future improvements to the filtering code.

As a result of this integration work, dummynet now supports multiple pipes, each of them configurable independently from the others. Traffic is filtered according to the rules of the `ipfw` packet filter, and selected packets can be diverted to different pipes. The `ipfw` rules allow a packet to be analysed multiple times, so that arbitrary topologies of pipes can be constructed if needed.

2.3 Configuration

Configuration of dummynet is done using the `ipfw` command, which has been extended to allow configuration of dummynet. A new filtering rule has been added:

```
ipfw add R pipe N ...
```

to forward matching packets to the specified pipe (multiple rules can point to the same pipe). Each pipe has a unique identifier (a 16-bit number). Configuration of pipes is done using commands starting with `ipfw pipe ...`, e.g.

```
ipfw pipe config N bw 100 delay 400 size 30
```

sets the parameters for pipe N to 100 Kbit/s, 400 ms, 30 buffers, whereas

```
ipfw pipe list
```

shows the configuration of currently defined pipes.

Typically, to make machine X appear behind a bottleneck link, one can configure a couple of pipes (one for each direction). It is not necessary that the pipes have the same features, e.g. the following commands

```
ipfw add 1000 pipe 0 ip from any to X in
ipfw add 1010 pipe 1 ip from X to any out

ipfw pipe config 0 bw 100 delay 40
ipfw pipe config 1 bw 10 delay 100
```

simulate the behaviour of an asymmetric channel between node X and the rest of the network.

Thanks to the packet filter, one can apply delays and bandwidth limitations to part of the traffic, while letting other traffic (e.g. NFS) run at full speed. In this way, a diskless machine (like the one where dummynet was initially developed and tested) can be used for experiments in a very convenient way.

Extensions to the tool, and to the configuration program, are relatively straightforward. As an example, one can add the already mentioned random losses and packet reordering, by adding parameters to set the distribution of such events. Or, random fluctuations of the bandwidth can be introduced to simulate the effect of competing traffic without actually having to generate it with some other application.

2.4 Applications

We implemented dummynet as a testing tool for evaluating protocol implementations with the flexibility of a simulator, and the simplicity of use of a real testbed. This motivates its location inside the kernel, and near the bottom of the protocol stack, so that all layers above it could be tested.

We believe that the tool fulfills its design goals. Being able to run experiments on a real implementation has several advantages, because it saves the need to build a model of the system for simulation purposes, and it can also spot implementation bugs which might go undetected otherwise. Operation on a single machine is also useful, because it makes the tool more convenient to use and also less subject to interference from other network traffic.

We find especially convenient the ability to run real applications, and to alter the features of the (simulated) network on the fly. With dummynet, questions like "how would this application work on a link with 128Kbit/s and 500ms delay" can be answered by setting up an experiment with a few keystrokes. This is even more important when the performance metrics are qualitative (e.g. user-perceived performance) rather than quantitative, and simulation could not produce useful results, while setting up a suitable testbed might be not feasible.

3 Forward Error Correction

Loss of packets is a fact of life in computer networks, be it due to communication errors, or simply to congestion phenomena. The usual approach to recover from losses is to retransmit missing packets on request (ARQ) [7]. There are however environments where this approach is not ideal. Classical examples come from the telecommunication world, where channels are often asymmetric or even unidirectional. Such channels are becoming more and more popular in computer networks, e.g. when data is transferred to mobile equipment or over a wireless channel.

ARQ has also limitations in multicast communications, because the possible lack of correlation between losses at different receivers can cause severe scalability problems, both in the amount of retransmitted traffic [5], and in the amount of feedback that the source has to manage [12]. In such situations, it might be convenient to devise an error recovery mechanism that can anticipate a certain amount of losses, and enable the receiver to recover all useful data without sending explicit requests for missing packets.

Such a mechanism can be implemented by applying a redundant encoding to the source data, so that even in presence of packet losses sufficient information is conveyed to the receiver to allow successful reconstruction of the original data. Such an encoding is called an *Erasure Code*.

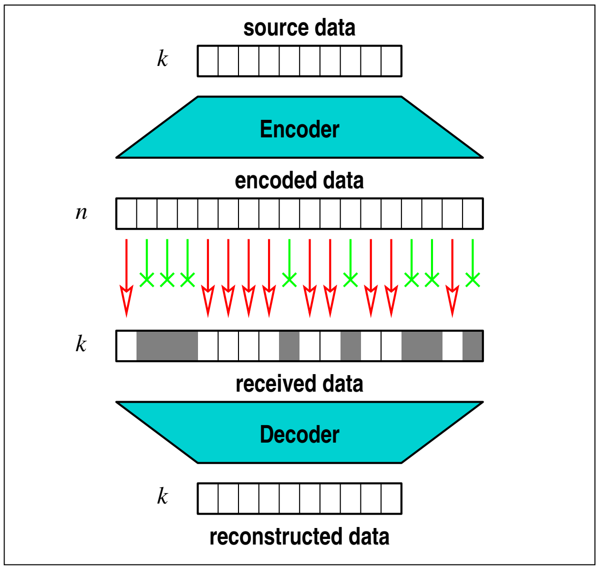

Figure 2: The principle of operation of an erasure code.

3.1 Erasure codes

The key idea behind an erasure code is to encode a set of k source data packets into a set of $n \geq k$ encoded data packets, in such a way that any subset of k encoded packets allows the reconstruction of the original sources (Fig. 2). Such a code is called an (n, k) erasure code, and can be used in several ways to recover from losses in a communication protocol, as it will be shown in Section 3.4. The issue now is how to produce the encoded packets given arbitrary values for k and n, and to make the encoding/decoding procedure sufficiently fast for practical use.

The problem is trivial to solve for special values of k and n. As an example, if $n = k + 1$, a simple parity computed over all packets will do the job. This is fast to compute, but can tolerate only a single loss per block of n packets. The case $k = 1$ is even simpler, because merely repeating packets solves the problem, but in this case the usage of the channel's capacity would be very bad.

3.2 A simple erasure code

For arbitrary k and n, a solution to build an erasure code comes from linear algebra: we know, in fact, that a polynomial of degree $k - 1$ is completely specified by its value in k different points. A simple encoding method is then to consider the source data packets (we can think of them as integer) as the *coefficients* of a polynomial P of degree $k - 1$, and construct the encoded packets as the values of P computed in n different points, e.g. $P(p_0), P(p_1) \ldots P(p_{n-1})$. At the receiver, standard procedures (e.g. see [13, Sec.2.8]) can be used to recover the coefficients of P given its values in k points. This code is called a *Vandermonde code* for reasons that will be clear in Section 3.2.3. There are, however, a few difficulties in implementing erasure codes (not only this one) in practice. We will describe them in the following sections, together with the way they can be solved.

3.2.1 High precision arithmetic

If packets are large, computations should be performed with high precision, and this is extremely slow because it is not supported natively by the architecture. This problem is relatively easy to solve, because we can split packets in smaller data units, e.g. bytes, and execute the same computations on each set of data units taken from different packets.

3.2.2 Operand expansion

But at this point we hit another significant problem, namely the expansion of operands: if the data (i.e. the coefficients of P) are represented on m bits, the values of $P(p_i)$ require a larger number of bits to be represented exactly if ordinary arithmetic is used. The expansion is roughly of $k \log n$ bits, which is extremely high considering that operands are typically small (32..64 bits at most). Luckily, also this problem can be solved, although the solution is trickier, by performing computations in a *Finite Field*, also called "Galois Field" or $GF(p^t)$ [2].

A *field* is a mathematical structure, defined by a set of field elements, and a sum and multiplication operation defined on them and satisfying certain properties. We are well acquainted with the property of a field, because the numbers we operate on everyday also constitute a field. Most of properties of linear algebra (including those related to polynomial interpolation) also apply to finite field computations. There exist finite fields with p^t elements, with p being a prime number.

A distinguishing feature of a Finite Field is that, unlike Integers or Reals, the number of field elements is finite. The finite size of the field is important for our purposes because, no matter what computations we do, we know how many bits we need at most to represent the results of our computations; in other words, there is no operand expansion.

One might wonder at this point how to do computations in a finite field, because the rules for sum and multiplication (see [2]) are complex at first sight. However, it turns out that for certain Finite Fields (e.g. when $p = 2$), the sum reduces to the exclusive OR, whereas multiplication can be implemented using a simple lookup table (because the field has a finite number of elements). This means that we can forget the intricacies of finite field computations, and assume that operations in a finite field have at most roughly the cost of a table lookup. More details on an efficient way to perform finite field computations are provided in [15].

3.2.3 Systematic codes

In some cases we would like the transmission to include a verbatim copy of the source data. Such an encoding is called *systematic*, and has the advantage that no decoding effort is necessary in absence of errors. The code proposed in Sec. 3.2 is not systematic, however it can be turned into a systematic code by simple algebraic manipulations, as follows.

We can look at the Vandermonde code by writing down the relation between the coefficients, x_i's, and the values of the polynomial, $y_j = P(p_j)$, as follows:

$$
\begin{aligned}
y_0 &= p_0^0 x_0 + p_0^1 x_1 + \ldots + p_0^{k-1} x_{k-1} \\
y_1 &= p_1^0 x_0 + p_1^1 x_1 + \ldots + p_1^{k-1} x_{k-1} \\
& \ldots \\
y_{n-1} &= p_{n-1}^0 x_0 + p_{n-1}^1 x_1 + \ldots + p_{n-1}^{k-1} x_{k-1}
\end{aligned}
$$

It turns out that the matrix, V, relating x_i's and y_i's is a Vandermonde matrix [13, Sec.3], and as such has the property that any minor of degree k extracted from V is invertible. This property still holds if we do linear combinations of the columns of V; thus, we can manipulate the matrix, using a Gaussian elimination procedure, in such a way that the upper k rows of the matrix become the identity matrix. After the transformation, we have that $y_i = x_i, i = 0 \ldots k-1$, which means that we have obtained a systematic code.

3.3 Performance

We have written a C implementation of the Vandermonde code described in this paper. Our code operates in $GF(2^t)$, the finite field with 2^t elements (with l ranging from 2 to 16). Field operations have been implemented using a lookup table for $t \leq 8$ (the table takes about 64 KB), or using the properties of logarithms for larger fields (in this case performance decreases by about a factor of 4). The performance data we provide in the following refer to $GF(2^8)$ which is a very convenient size for practical purposes[1].

Our code implements both encoding and decoding using a systematic matrix, operating on packets of user-defined size. It can produce redundancy packets one at a time, which is a convenient feature for some applications where the amount of redundancy is not fixed a-priori. The code is extremely compact and has a relatively small memory footprint, permitting its use even on small memory machines, and allowing a good interaction with cache memories.

The encoding and decoding speeds can be roughly expressed as c/l, where l is the number of non-source packets produced by the encoder, and the number of source packets missing at the decoder, respectively. The value of the constant c varies widely depending on the architecture, CPU speed, and especially, the memory bandwidth. Table 1 presents performance data for a number of different architectures, ranging from an HP100LX (a small 8086-based palmtop) to high end workstations using Pentium, Sparc and Alpha processors. From these results, we see that encoding/decoding can be done at high speed (with the constant c as high as 10 MBytes/s even on a Pentium133), thus suggesting that FEC-based error recovery can be implemented effectively on modern machines.

[1] the choice of the field size affects the parameter n of the code: recall from Sec. 3.2 that we compute the polynomial in n different points, so the field must have at least $p^t \geq n$ elements.

CPU	MHz	c (MBytes/s)
SPARC	167	21.2
SPARC	143	18.8
Alpha	255	12.6
PA7000	100	10.3
Pentium	133	9.6
i486	66	3.4
i386	25	.37
8086	8	.070

Table 1: Speed of operation of our decoder on different machines.

3.4 Applications

There are several application of an erasure code in computer communications. The first one that comes to mind is to use a pure FEC approach to improve the resilience to losses in unicast protocols: assuming we have to deliver k packets to a receiver, and we expect some amount of losses, we can send unconditionally a suitable number $n > k$ of packets that guarantees successful reception of at least k packets with high probability.

By using the same approach, and a multicast communication infrastructure, we can easily accommodate multiple receivers for the same data, even in presence of different or uncorrelated loss patterns at the different receivers. All we have to do is to tune the amount of redundancy according to the features of the worst receiver.

The scalability of reliable multicast protocols based on FEC is often much better than those based on ARQ for loss recovery. As an example, Figure 3 shows the average number of transmissions for each packet in a multicast application where receivers are subject to random independent losses, with ARQ or with FEC and different values of k. The advantages of using FEC, even with moderate blocksizes or small number of receivers, is evident from the graph. A more detailed discussion of scalability issues is present in [11].

Pure FEC is interesting in that it requires no feedback at all from the receivers, but has the drawback of adapting badly to variable network conditions. In these cases, a hybrid FEC+ARQ approach can be used, which consists in using only a moderate amount of redundancy by default (or, possibly, even no redundancy at all), and sending additional *repair packets* on demand. Compared to pure ARQ, the important difference is that all packets are equivalent for recovery purpose, and this is also true in presence of multiple receivers. As a consequence, the handling of feedback is highly simplified, because the receiver does not need to specify *which* packets are missing, but only *how many* of them are missing. This also increases the chance that retransmission requests from different receivers can be merged, thus alleviating some scalability problems which might exist in multicast communication protocols.

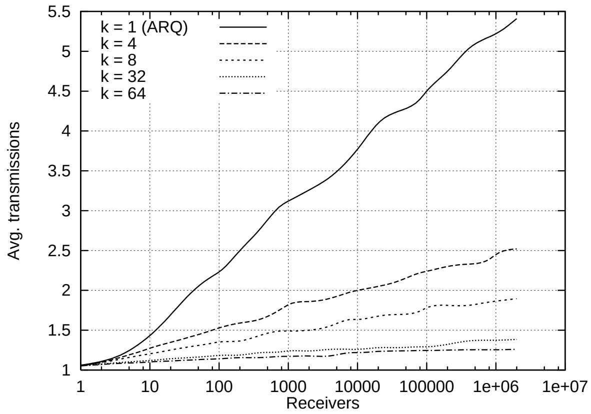

Figure 3: Number of transmissions per packet for a multicast protocol in presence of 5% losses, with ARQ and with FEC.

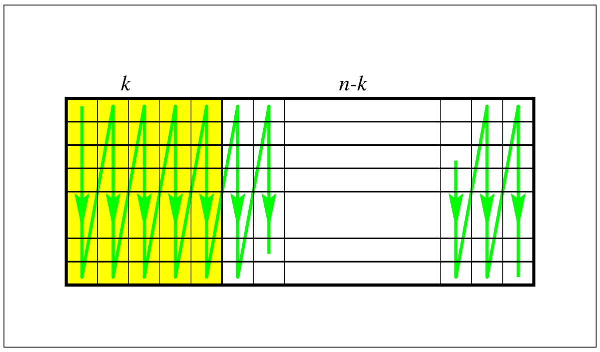

Figure 4: Transmission order in RMDP.

An example of such a protocol is RMDP [16], which is the application for which the Vandermonde code presented in this paper was originally developed.

RMDP is a multicast file-transfer application operating on top of IP multicast, or on wireless channels were receivers have limited uplink communication capabilities. Thanks to the FEC encoding, the protocol scales very well even in presence of highly variable loss patterns at the receivers, or differences in receive bandwidths.

In RMDP, a server accepts requests for a file from one (or more) receivers, and then transmits the file as UDP packets over multicast IP [3]. The file is partitioned in slices of k packets each, and these packets are passed to an encoder which can produce up to n different encoded packets. The transmission occurs by picking one (encoded) packet per slice (see Fig. 4). Receivers with missing packets (or late comers) issue "continuation" requests, asking for more packets. The transmitter responds by generating a new packet for each slice using the encoder, and sending them. This way, each receiver can complete reception by accumulating a sufficient number of packets per slice, no matter which ones.

More recently, we have proposed in [18] the use of FEC as a mechanism to support layered multicast congestion control in reliable multicast communication, using techniques formerly devised only for the (unreliable) transfer of continuous streams. Layered congestion control relies on the transmission of a data stream over a set of multicast channels, or *layers* [10]. Data is encoded in a hierarchical fashion, so that receivers with low bandwidth subscribe only to the lowest layer (getting a low-quality version of the stream), whereas high-bandwidth receiver can get a higher quality stream by subscribing to more layers. A congestion control mechanism drives the process of joining/leaves layers by looking at the loss patterns experienced by each receiver. We have adapted this mechanism to reliable multicast by using an encoder capable of producing a suitably large number of packets ($n \gg k$), and transmitting different (encoded) packets across all layers. Thanks to the encoding, the effect of hopping between layers does not affect the efficiency of the data transfer, because again it is the total number of received packets, not their identity, that counts.

The erasure code presented in this paper has been in use since early 1997 in a number of research papers and actual applications (e.g. [11, 17]) for the development of scalable multicast communication protocols. Albeit some other erasure codes with better asymptotic performance have been proposed [8] recently, we believe that our code has many practical applications being deterministically decodable, small and simple (we have run it on machines as small as an HP100LX), and, especially, not subject to patents or other impediments to its use.

4 Conclusions

We have presented a couple of tools that can be useful in the design, development and analysis of networking protocols and applications. Both tools are available in source format from the author's Web page [2] together with with more detailed documentation. While developed on FreeBSD, one design goal was to make the code easily portable to different operating systems, so that we have avoided the use of special operating systems features if not strictly necessary (in particular, in dummynet we did not use the queue handling macros which are present in the operating system sources, and we split the core functionalities from the packet filter). The small size of the sources, both for dummynet and for the Vandermonde code, is another factor that should make the porting effort relatively simple.

Acknowledgements

This work has been partly supported by the Ministero dell' Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the Project "Design Methodologies and Tools of High Performance Systems for Distributed Applications".

References

[1] A.Albanese, J.Bloemer, J.Edmonds, M.Luby, M.Sudan, "Priority Encoding Transmission", 35th Annual Symposium on Foundations of Computer Science, IEEE Computer Science Press, 1994.

[2] R.E.Blahut, "Theory and Practice of Error Control Codes" Addison Wesley, MA, 1984.

[3] S.Deering, "Multicast Routing in a Datagram Internetwork", PhD Thesis, Stanford University, Dec.1991.

[2] `http://www.iet.unipi.it/~luigi/`

[4] A.Heybey, "The network simulator", Technical Report, MIT, Sept.1990

[5] C.Huitema, "The Case for packet level FEC", Proc. 5th Workshop on Protocols for High Speed Networks, pp.109-120, Sophia Antipolis, France, Oct.1996.

[6] S.Keshav, "REAL: A Network Simulator", Technical Report 88/472, Dept. of Computer Science, UC Berkeley, 1988. `http://netlib.att.com/~keshav/papers/real.ps.Z`, `ftp://ftp.research.att.com/dist/qos/REAL.tar`

[7] S.Lin, D.J.Costello, M.Miller, "Automatic-repeat-request error-control schemes", IEEE Comm. Magazine, v.22,n.12, pp.5-17, Dec.1984

[8] M.Luby, M.Mitzenmacher, A.Shokrollahi, D.Spielman, and V.Stemann. "Practical loss-resilient codes", Proc. of the Twenty-Ninth Annual ACM Symp. on Theory of Computing, El Paso, Texas, 4-6 May 1997.

[9] S.McCanne, S.Floyd, ns-LBNL Network Simulator (`http://www-nrg.ee.lbl.gov/ns/`)

[10] S. McCanne, V. Jacobson, and M. Vetterli, "Receiver-driven Layered Multicast", ACM SIGCOMM'96, August 1996, Stanford, CA, pp.1-14.

[11] J. Nonnenmacher, E.W.Biersack, D.Towsley, "Parity-Based Loss Recovery for Reliable Multicast Transmission", SIGCOMM'97, Cannes, France, 14-18 Sep.1997.

[12] J. Nonnenmacher, E.W.Biersack, "Optimal Multicast Feedback", Proc. of INFOCOM'98, S.Francisco, Mar.29-Apr.2 1998, IEEE.

[13] "Numerical Recipes in C: the Art of Scientific Computing", Cambridge University Press.

[14] L.Rizzo, "Dummynet: a simple approach to the evaluation of network protocols", ACM Computer Communication Review, Vol.27, n.1, January 1997, pp.31-41.

[15] L. Rizzo, "Effective erasure codes for reliable computer communication protocols", ACM Computer Communication Review, Vol.27, n.2, April 1997, pp.24-36. `http://www.iet.unipi.it/~luigi/fec.ps`

[16] L.Rizzo, L.Vicisano, "RMDP: an FEC-based Reliable Multicast protocol for wireless environments", ACM Mobile Computing and Communications Review, Vol.2, n.2, April 1998.

[17] E.Schooler, J.Gemmel, "Using Multicast FEC to Solve the Midnight Madness Problem", Microsoft Research Tech. Report MSR-TR-97-25.

[18] L.Vicisano, L.Rizzo, J.Crowcroft, "TCP-like congestion control for layered multicast data transfer", Proc. of INFOCOM'98, S.Francisco, Mar.29-Apr.2 1998, IEEE.