Chapter 7
Application of Proposed FEC Schemes in TCP/IP using NS-2

7.1 Introduction
This chapter explains the implementation of CDMA, OFDM, CDMA-OFDM systems in Network simulator-2. It also discusses implementation of the proposed schemes in NS-2. Results are obtained in terms of packet error rate, from the simulation.

7.2 Introduction to Network Simulator-2 (NS-2)

Network Simulator (NS-2) is a discrete event simulator targeted at networking research. NS provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks. NS-2 is an object oriented simulator, written in C++, with an OTcl interpreter as a frontend. The simulator supports a class hierarchy in C++ (also called the compiled hierarchy in this document), and a similar class hierarchy within the OTcl interpreter (also called the interpreted hierarchy in this document). The two hierarchies are closely related to each other; from the user's perspective, there is a one-to-one correspondence between a class in the interpreted hierarchy and one in the compiled hierarchy. The root of this hierarchy is the class Tcl Object. Users create new simulator objects through the interpreter; these objects are instantiated within the interpreter, and are closely mirrored by a corresponding object in the compiled hierarchy. The interpreted class hierarchy is automatically established through methods defined in the class Tcl Class. User instantiated objects are mirrored through methods defined in the class Tcl Object. There are
other hierarchies in the C++ code and OTcl scripts; these other hierarchies are not mirrored in the manner of Tcl Object.

The Simulator uses two languages because simulator has different functions to perform. On one hand, detailed simulations of protocols require a systems programming language which can efficiently manipulate bytes, packet headers, and implement algorithms that run over large data sets. For these tasks run-time speed is important and turn-around time (run simulation, find bug, fix bug, recompile, re-run) is less important. On the other hand, a large part of network research involves slightly varying parameters or configurations, or quickly exploring a number of scenarios. In these cases, iteration time (change the model and re-run) is more important. Since configuration runs once (at the beginning of the simulation), run-time of this part of the task is less important. NS meets both of these needs with two languages, C++ and OTcl. C++ is fast to run but slower to change, making it suitable for detailed protocol implementation. OTcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration. NS (via tcl) provides glue to make objects and variables appear on both languages.

The Class Agent
The class Agent has an implementation partly in OTcl and partly in C++. The C++ implementation is contained in the files ~ns/agent.cc and ~ns/agent.h, and the OTcl support is in ~ns/tcl/lib/ns-agent.tcl. The C++ class Agent includes enough internal state to assign various fields to a simulated packet before it is sent. This state includes the following:
addr_ node address of source address in packets
dst_ Destination where packets to get received
size_ packet size in bytes (placed into the common packet header)
type_ type of packet (in the common header, see packet.h)
fid_ the IP flow identifier (formerly class in ns-1)
prio_ the IP priority field
flags_ packet flags (similar to ns-1)
defttl_ default IP ttl value

These variables may be modified by any class derived from Agent, although not all of them may be needed by any particular agent.

Agent methods

The class Agent supports packet generation and reception. The following member functions are implemented by the C++ Agent class, and are generally not over-ridden by derived classes:

Packet* allocpkt() allocate new packet and assign its fields
Packet* allocpkt(int) allocate new packet with a data payload of n bytes and assign its fields.

The following member functions are also defined by the class Agent, but are intended to be over-ridden by classes deriving from Agent:

void timeout(timeout number) subclass-specific time out method
void recv(Packet*, Handler*) receiving agent main receive path

The allocpkt() method is used by derived classes to create packets to send. The function fills in the following fields in the common packet header (Section 7.3): uid, ptype, size, and the following fields in the IP header: src, dst, flowid, prio, ttl. It also zero-fills in the following fields of the Flags header: ecn, pri, usr1, usr2.

Any packet header [63] information not included in these lists must be handled in the classes derived from Agent. The recv() method is the main entry point for an
Agent which receives packets, and is invoked by upstream nodes when sending a packet. In most cases, Agents make no use of the second argument (the handler defined by upstream nodes).

Packet Headers and Formats

The procedures and functions described in this section are present in the source code files \texttt{-ns/tcl/lib/ns-lib.tcl}, \texttt{-ns/tcl/lib/ns-packet.tcl}, and \texttt{-ns/packet.{cc, h}}. Objects in the class Packet are the fundamental unit of exchange between objects in the simulation. The class Packet provides enough information to link a packet on to a list (i.e., in a Packet Queue or on a free list of packets), refer to a buffer containing packet headers that are defined on a per-protocol basis, and to refer to a buffer of packet data. New protocols may define their own packet headers or may extend existing headers with additional fields. New packet headers are introduced into the simulator by defining a C++ structure with the needed fields, defining a static class to provide OTcl linkage, and then modifying some of the simulator initialization code to assign a byte offset in each packet where the new header is to be located relative to others.

Packet Classes

There are four C++ classes relevant to the handling of packets and packet headers in general: Packet, p\_info, PacketHeader and PacketHeaderManager.

- The class Packet defines the type for all packets in the simulation; it is a subclass of Event so that packets may be scheduled (e.g. for later arrival at some queue).
- The class packet\_info holds all text representations for packet names.
- The class PacketHeader provides a base class for any packet header configured into the simulation. It essentially provides
enough internal state to locate any particular packet header in the
collection of packet headers present in any given packet.

- The class PacketHeaderManager defines a class used to collect
  and manage currently-configured headers. It is invoked by a
  method available to OTcl at simulation configuration time to enable
  some subset of the compiled-in packet headers.

The Packet Class

The class Packet defines the structure of a packet and provides member
functions to handle a free list for objects of this type. It is illustrated in Figure 7.1
and defined as follows in the header file ‘packet.h’, as follows.

    class Packet : public Event {
    private:
        friend class PacketQueue;
        u_char* bits_;  
        u_char* data_; /* variable size buffer for 'data'*/
        u_int datalen_; /* length of variable size buffer */
    protected:
        static Packet* free_; 
    public:
        Packet* next_; /* for queues and the free list */
        static int hdrlen_; 
        Packet() : bits_(0), datalen_(0), next_(0) {} 
        u_char* const bits() { return (bits_); } 
        Packet* copy() const; 
        static Packet* alloc(); 
        static Packet* alloc(int); 
        inline void allocdata(int); 
        static void free(Packet*); 
        inline u_char* access(int off) {

159
if (off < 0)
abortQ;
return (&bits_[off]);
}
inline u_char* accessdata() { return data_; }
};

This class holds a pointer to a generic array of unsigned characters (commonly called the “bag of bits” or BOB for short) where packet header fields are stored. It also holds a pointer to packet “data” (which is often not used in simulations). The bits_ variable contains the address of the first byte of the BOB. Effectively BOB is (currently implemented as) a concatenation of all the structures defined for each packet header that have been configured in. BOB generally remains a fixed size throughout a simulation, and the size is recorded in the Packet::hdrlen_ member variable. This size is updated during simulator configuration by OTcl1. The other methods of the class Packet are for creating new packets and storing old (unused) ones on a private free list. Such allocation and deallocation is performed by the following code (in ~ns/packet.h):
7.3 Modulation in NS-2

In the physical layer of the simulator, BPSK is employed in the header file `~ns\modulation.h` and `~ns\modulation.cc` files. The modulation is also implemented as an object whose class is defined as follows:

```cpp
class Modulation {
    friend class NetIf;

public:
    virtual int BitError(double) = 0; // success reception?

protected:
    int Rs; // symbol rate per second

private: // Probability of 1 bit error
    virtual double ProbBitError(double) = 0;
    // Probability of n bit errors
};
```
virtual double ProbBitError(double, int) = 0;
};
class BPSK : public Modulation {
public:
    BPSK(void); /* ofdm(void)
    BPSK(int);
    virtual int BitError(double Pr);
private:
    virtual double ProbBitError(double Pr);
    virtual double ProbBitError(double Pr, int n);
};

7.4 Implementation of CDMA, OFDM and CDMA-OFDM systems on Rayleigh channel in NS-2

The basic unit of data in NS-2 is in the form of packet which is going to be sent through the physical layer for transmission to receiver. Here it is needed to access the packet information in order to modulate and send in CDMA or OFDM or CDMA-OFDM mode. In NS-2, the packet is implemented as an object which is going to be passed between the Agents. Agents represent endpoints where network-layer packets are constructed or consumed, and are used in the implementation of protocols at various layers. The inbuilt BPSK scheme in the Network Simulator is changed to CDMA or OFDM or CDMA-OFDM system [71] [76] [77] with QPSK as modulation scheme. The Matlab programs of the same are converted to the C code by using the Matlab compiler, which generates 3 files the header file cdma.h, cdma.c, and cdma_main for CDMA implementation ofdm.h, ofdm.c, and ofdm_main for OFDM implementation and cdma-ofdm.h, cdma-ofdm.c, and cdma-ofdm_main for CDMA-OFDM implementation. This code is again modified for the variable names compatible with the NS-2's C++ code variables. These have been called from the modulation function which is shown as commented in the code.
7.5 Implementation of Concatenated FEC Schemes in NS-2

The Matlab code of proposed FEC is implemented in fec.h and fec.c in the generalized FEC model class. The structure of FEC model is as follows.

```cpp
static class FECModelClass : public TclClass {
public:
    FECModelClass() : TclClass("FECModel") {}
    TclObject* create(int, const char* const*) { return (new FECModel); }
}

class_fecmodel; FECModel::FECModel() : firstTime_(1), FECstrength_(1) {
    int FECModel::command(int argc, const char* const* argv)
    {
        Tcl& tcl = Tcl::instance();
        if (argc == 3) {
            if (strcmp(argv[1], "FECstrength") == 0) {
                FECstrength_ = atoi(argv[2]);
                return (TCL_OK);
            }
        } else if (argc == 2) {
            if (strcmp(argv[1], "FECstrength") == 0) {
                tcl.resultf("%d", FECstrength_);
                return (TCL_OK);
            }
        }
        return BiConnector::command(argc, argv);
    }

    int fix_ = 0;
    void FECModel::recv(Packet* p, Handler* h)
    {
        hdr_cmn* ch = hdr_cmn::access(p);
        if (ch->direction() == hdr_cmn::DOWN) {
            addfec(p);
            downtarget_->recv(p, h);
            return;
        } else {
            if (ch->errbitcnt() > FECstrength_)
            {
                Packet::free(p);
            } else if (ch->errbitcnt() && (ch->errbitcnt() <= FECstrength_)) {
                ch->errbitcnt() = 0;
                ch->error() = 0;
                printf("FEC: \d fixed\n", fix_++);
                subfec(p);
                uptarget_->recv(p, h);
            }
        }
    }

    void FECModel::addfec(Packet* p)
    {
```
int bit = hdr_cmn::access(p)->size() * 8;
FECbyte_ = (int)(log(bit) / log(2));
FECbyte_ *= FECstrength_;
FECbyte_ = (FECbyte_ + 7) / 8;
//  firstTime_ = 0;
// printf("FEC Down = %d \n", hdr_cmn::access(p)->size());
hdr_cmn::access(p)->size() += FECbyte_;  
hdr_cmn::access(p)->fecsize() = FECbyte_;}

void FECModel::subfec(Packet* p)
{    FECbyte_ = hdr_cmn::access(p)->fecsize();
    hdr_cmn::access(p)->size() -= FECbyte_; 
    // printf("FEC Up = %d \n", hdr_cmn::access(p)->size());}

The Matlab program of Proposed concatenated FEC scheme-I and Proposed concatenated FEC scheme-II are compiled to form FEC.h and FEC.cc file of NS-2. To implement proposed FEC in CDMA or OFDM or CDMA-OFDM systems, appropriate header file should be properly linked and reinstall ns-2.

7.6 NS-2 Simulation Results

The simulation is done for 50 seconds, for a three node wireless network having the nodes (0,1,2) shown as in figure 7.2. The nodes start moving from 0.5 sec onwards, away from each other. From the time instant \( t = 3.000 \) sec onwards, node-0 starts to send packets to node-1 and as both the nodes are with in the hearing distance, the connection is established and it can be seen that the line between node-0 to node-1 in the Figure 7.2 shown below.

As the time progresses, the two nodes getting away from each other and at time instant \( t=19.300 \) sec onwards, they lost direct-connection. At the same time the
link is established via node-2 which starts forwarding the packets from node-0 to node-1 and it can be seen that two connections in the Figure from \( t = 19.300 \) sec onwards. Still the node was traveling to move away from node-0 and the link is through node-2 only.

Finally from time \( t = 41.600 \) sec onwards the lines are not seen between the nodes, meaning that whatever the packets being sent were lost as the links got broken. At \( t = 50.000 \) the simulation stops. The following table shows the node positions and the links between the various nodes at different instants.

As soon as transmission by simulation is over, the status of packet transmitted from source node to destination node can be viewed with the help of Trace Graph. This trace Graph directly displays the results in Matlab window.
Figure 7.2 Simulation scenarios of CDMA with proposed FEC Scheme-I
The simulation by NS-2 gives two types of packet losses namely lost packets and dropped packets. The dropped packets are the ones which have not properly reached the destination, even though a link is being established between the communicating nodes. The lost packets are the ones which are ignored by the receiver after checking its BER in the packet. NS-2 simulator which has built-in TCP/IP suite. The Physical layer TCP is modified to have CDMA or OFDM or CDMA-OFDM transceiver structure and transport layer of TCP is also modified to FEC scheme-I and scheme-II. For these schemes, the performance of TCP is simulated and studied. The dropped packet rate, lost packet rate and packet error rate are as shown in Figures 7.3, 7.4 and 7.5.

From the Figure 7.3, it is observed that the dropped packet rate in conventional TCP system is higher. The dropped packet rate is decreased very much, if the physical layer of TCP is modified to CDMA, OFDM and CDMA-OFDM with FEC scheme-I.

The lost packet rate performance is shown in Figure 7.4. The lost packet rate is also getting decreased by very much when compared to the performance of conventional TCP.
Figure 7.3 Comparison of Dropped Packets Error Rate in CDMA, OFDM and CDMA-OFDM systems having Proposed FEC Scheme-I in Rayleigh channel

Figure 7.4 Comparison of Lost Packets in CDMA, OFDM and CDMA-OFDM systems having Proposed FEC Scheme-I in Rayleigh channel
The overall packet error rate in TCP is tremendously decreased if TCP physical is modified and FEC scheme-1 is implemented. The same is represented in Figure 7.5.

Figure 7.5 Comparison of Overall Packet Error rate in CDMA, OFDM and CDMA-OFDM systems having Proposed FEC Scheme-I in Rayleigh channel

7.8 Performance of the Novel Concatenated FEC Scheme –II

The proposed FEC scheme by concatenation of Turbo and Irregular Turbo is implemented in transport layer of TCP and Physical layer is modified to have CDMA or OFDM OR CDMA-OFDM.

From Figure 7.6, it is observed that the dropped packet rate in conventional TCP system is higher. The dropped packet rate is decreased very much, if the
physical layer of TCP is modified to CDMA, OFDM and CDMA-OFDM with FEC scheme-1.

The lost packet rate performance is shown in Figure 7.7. The lost packet rate is also getting decreased considerably when compare to the performance of conventional TCP.

![Dropped packet error rate in Rayleigh Channel](image)

**Figure 7.6** Comparison of Dropped Packets CDMA, OFDM and CDMA-OFDM systems having Proposed FEC Scheme-II in Rayleigh channel
The Figure 7.8 shows the performance of TCP with CDMA or OFDM or CDMA-OFDM. The overall packet error rate in TCP is tremendously decreased if TCP physical layer is modified and FEC scheme –II is implemented.
Comparison of Overall packet rate in rayleigh Channel

Conventional TCP transmission with out FEC
CDMA with Turbo & Ir. Turbo code(10%)
OFDM with Turbo & Ir. Turbo code(10%)
CDMA-OFDM with Turbo & Ir. Turbo code(10%)

Figure 7.8 Comparison of Overall Packet Error Rate in CDMA, OFDM and CDMA-OFDM systems having proposed FEC Scheme-II in Rayleigh channel

7.9 Conclusions

It is observed from NS-2 simulations that CDMA, OFDM and CDMA-OFDM systems with the proposed concatenated FEC schemes outperforms very well in TCP suite. Hence these codes can be employed as FEC code for future generation wireless networks.