REFERENCE MANUAL
ns-3 Wireless Channel Model based on a Hidden Makov Process

David Gómez Fernández and Ramón Agüero Calvo

24th April 2012

1 Channel model based on a Hidden Markov Process

According to [1]: "A hidden Markov process is a discrete-time finite-state homogeneous Markov chain observed through a discrete-time memoryless invariant channel. The channel is characterized by a finite set of transition densities indexed by the number of states of the Markov chain".

A Hidden Markov Process (*HMP*) model is based on a system with N independent states (S_i, where i is the index of the corresponding state). The transition between the different states is established by a number of stochastic probabilities. These (let us name them transition probabilities) are represented as $a_{i,j}$, where the tuple (i, j) represents both the current state (i) and the next one (j). In order to fully define the process, we also need another set of probabilities, which are used for decision purposes, associated with each output value on the system; every of these values is defined as $b_i(k)$, where i refers to the current state, and k defines the corresponding output symbol. It is worth mentioning that in these models, unlike the legacy *Gilbert-Elliot*, these states are "hidden", and therefore each state is not one-to-one mapped to a specific output. Last, but not least, we need to define the initial state for the system; for that purpose, the vector π_i sets the probability of being in the state i at the moment the system gets started.

Taking into account how the model is implemented, we are able to define a complete *HMP* channel by means of the following elements:

1. Number of states in the model[1], N.
2. Number of output values, M (in this work, there will be only two possible outputs: correct o erroneous reception).
3. Transition matrix (A), with dimension $N \times N$, containing the whole set of state transition probabilities, $a_{i,j}$.
4. Decision matrix (B), with dimension $N \times M$, which contains the probabilities for all the possible outputs at each state $b_i(k)$.
5. The initial probability distribution of being on each state, $\Pi = \{\pi_i\}$.

In order to complete the model configuration, we need to estimate the *probability density function* (*pdf*) of the time spent at a particular state i. It is well-known that such *pdf* follows a negative exponential distribution, $f_{T_i}(t_i) = \lambda_i \cdot e^{-\lambda_i \cdot t_i}$, with $\overline{t_i} = \frac{1}{\lambda_i}$ as the average time spent at state i. In this case, $\overline{t_i}$ is estimated based on the average number of consecutive frames for each state, as shown in Eq. 1.

[1]In this work we will only discuss, due to space restrictions, the results obtained from a *4-state* Hidden Markov Process.

Average consecutive frames at state $i = \overline{N_i} =$

$$= \sum_{i=0}^{\infty} n \cdot p_i(n) = \sum_{i=0}^{\infty} n \cdot a_{i,i}^{n-1} \cdot (1 - a_{i,i}) = \frac{1}{1 - a_{i,i}} \quad (1)$$

Finally we can get (Eq. 2) the value of $\overline{t_i}$, where ψ denotes the average inter-frame duration:

$$\overline{t_i} = \psi \cdot \overline{N_i} = \frac{\psi}{1 - a_{i,i}} \quad (2)$$

One of the most relevant (and novel) aspects of this work is the configuration of the corresponding Hidden Markov Process based on time units, rather than on frames. This is of outer relevance, since in real protocols, it is usually hard to ensure fixed intervals between consecutive frames (for example TCP congestion control algorithms halts the transmitter for certain amount of time upon certain circumstances). In order to address this requirement, we start from a set of measurements carried out with UDP traffic, where the source nodes transmits as fast as the channel is able to support (saturation situation). From these measurements, summarized in Table 1, we have trained (with the `hmmtrain` Matlab function, limiting the number of iterations to 1000), three illustrative scenarios, ranging from a **Bad** channel (measurement #5), with quite negative transmission conditions to a **Good** channel (measurement #12), with a performance closer to that of an error-free transmission; we also selected an **Average** channel (measurement #9), representing a reasonably average operation. The training is done at a frame level but, since the channel is saturated, it can be easily changed to a time-based configuration by multiplying with the average frame duration[2], which will be used during the subsequent simulation campaign.

Before going through the rest of the document, we **must remark that this model is only valid in a IEEE 802.11b link**, because the characterization was carried out by means of a real measurement campaign over the aforementioned amendment.

2 Simulation configuration

In this section we will try to showcase the most highlighting stuff that the Hidden Markov Process model supports. It is worth mentioning that we have strictly followed the `ns-3` coding style (please take a look to [2]) recommended by the official website [3], which follows at the same time the GNU coding standard [4].

[2]In 802.11b this is approximately 2 ms.

Table 1: UDP behavior over a real IEEE 802.11b indoor lossy link

#	***Thput*** *[Mbps]*	***FER***	***PER***	***EFB*** *Average*	*Maximum*	*Variance*
1	0.82	0.814	0.5	15.975	2759	9586.49
2	1.34	0.709	0.314	5.929	1035	740.949
3	1.49	0.676	0.297	7.502	1229	1675.485
4	2.32	0.53	0.146	3.644	1927	1478.843
5	2.33	0.517	0.179	6.217	821	983.664
6	2.72	0.465	0.105	3.014	383	166.981
7	3.58	0.331	0.058	2.6	258	79.528
8	3.76	0.301	0.059	3.408	259	138.689
9	3.80	0.298	0.127	4.836	219	301.49
10	4.00	0.268	0.044	2.767	320	134.182
11	4.04	0.261	0.05	3.065	321	221.041
12	4.79	0.163	0.025	2.633	144	57.627
13	5.50	0.069	0.012	3.136	75	76.007
14	5.96	0.014	0.002	2.84	16	12.854
15	5.99	0.013	0.001	1.361	7	0.932

2.1 Class definition

First of all, we must remark that this model **do not have any kind of dependency with the signal quality** (e.g. distance between nodes, SNR...) In such a case, we have tried to mimic the bursty nature of the wireless channel by means of "training" a hidden Markov process from 15 different measurements[3] obtained over a empirical campaign carried out over a real indoor office-like scenario (see Table 1, where the worst case transmission corresponds to the measurement #1 and the best one is represented in #15).

We will show below the different transition and decision matrices for every measurement "trained" (15 possibilities), but before that, it worth highlighting that we have 6 types of configurations for each type of channel, focusing on the 4-state model[4]:

1. Total freedom. This belongs to a special case in which a state could change to another without any kind of restriction.
2. $a(0,3) = a(3,0) = 0$. This case forbids the direct transition between the extreme states.
3. Birth-death process. A state can only change to a contiguous neighbour.
4. $b_0(1) = b_3(0) = 0$. This means that in the extreme states the probability of having an opposite decision (correct reception in the bad state, and vice versa) is null.

[3]With these ones we think we trustworthy span all the range of possible results in a real transmission, that is to say, from situations in which a channel behaves almost as an ideal one to the opposite extreme cases which yield a frame error rate close to a 55%

[4]In order to avoid repetitive comments, in this manual we will only focus on this particular 4-state Markov chain.

5. Combination of 2 and 4.
6. Combination of 3 and 4.

Regarding the programming-related issues, we have defined the following list of attributes for the class `HiddenMarkovErrorModel`:

- `RanVar`. Random variable that will help to decide whether a packet is successfully received or not. By default, it is set to an `UniformVariable` which spans a range between $[0, 1]$.
- `AverageFrameDuration`. Average time (in microseconds) between two consecutive frames (used to model the exponential inter arrival process). The parameter is tightly linked with time-basis behaviour, yielding the mean value in which the chain stays in a particular state: namely, it is defined as the product between this value and the average sojourn time that a concrete state i stays
- `InitialState`. Set the state in which the simulation starts. The default value is 0.
- `DecisionStates`. Number of system outputs. For this hidden Markov process, designed for mimicking the behavior of a real IEEE 802.11b channel, we have only two possibilities: a correct reception or a frame loss (due to the propagation, possible collisions...). Hence, the value configured by default is 2.
- `ErrorUnit`. The *HMP* model is able to perform two types of transition triggers: the first one tackles the state change after the reception of each frame[5]; on the other hand, we have carried out a rather different character, where the most relevant parameter focuses on the timing between the frame reception (in this case, we model the average time in a concrete state i with a exponential random variable, as mentioned in Section 1), that is to say,
- `TransitionMatrixFileName`. Relative path (from the folder ns-3-dev[6]). The file contains the state matrix (NxN), being N the number of "hidden" states), holding the whole set of state transition probabilities, $a_{i,j}$.
- `EmissionMatrixFileName`. Relative path to the file that contains the probabilities for all the possible outputs at each state $b_i(k)$. Remark once again that we do have two states in order to model a wireless channel: the state "0" represent that the frame is received with errors and must be retransmitted (if and only if the transmitter node has not reached the maximum number of IEEE 802.11 retransmissions allowed); otherwise, the state "1" showcases a successful reception[7].
- `HiddenMarkovErrorModelRxTrace`. Frame tracing hook.

[5]This type of analysis might lead to incorrect results, because it supposes a constant interval between two consecutive frames, behavior undoubtedly incorrect when the channel introduces frame losses into the system.

[6]By default, the configuration files are located into the following path: */src/hidden-markov-error-model/configs/*.

[7]These configuration files implicitly define the number of hidden states (N) of the Markov chain.

2.2 Testing

We will show below some examples illustrating how to test the behaviour of this channel model. Namely, the code in charge of testing the different wireless models (recall that **they are only valid for IEEE 802.11b links at 11 Mbps**[8]) is located in the `scratch` folder (you can change it wherever you want), whose file is named `error-model-test`. Through this piece of code we can configure the simulation scenario by means of the following parameters:

- `ChannelModel`. This parameter allows to define the wireless link type[9] between the source and the sink nodes. We will set the parameter as "HMM" if we want to make use of this model.
- `HmmChannel`. Select either the *"Good"*, *"Average"* or *"Bad"* channel configuration. This parameter acts as a simple shortcut, since it links both the `EmissionFileHmm` and the `TransitionFileHmm` attributes to measurements #12, #9 and #5, respectively (check Table 1 for further details).
- `TransitionFileHmm`. Relative path to the file which contains the transition matrix (by default: `HMM_4states/HMM_03_TR_1.txt`).
- `EmissionFileHmm`. As the parameter above, this one specifies the name of the emission matrix which will be held by the wireless channel (by default: `HMM_4states/HMM_03_TR_1.txt`).
- `PacketLength`. This value configures the packet size. By default, it is set to 1472 bytes (i.e. typical maximum size for an UDP datagram over Ethernet).
- `InterPacketTime`. Time gap between two consecutive packets (at application level).
- `Distance`. We can tweak the distance between the source and the sink nodes. It is worth remarking again that this parameter does not affect a channel model based on a *HMP*, so it is irrelevant in this case.
- `TransportProtocol`. We can choose either TCP or UDP as the transport layer

Please take into account that this piece of code does not pretend to be an elegant solution to test the channel model; it is just a naive compilation of ideas that aims to quantitatively characterize the different error model we have carried out.

3 Tracing

In the model we have opted for making our own proprietary trace system, in order to control the whole output environment. For this purpose, we have

[8]Since *BEAR* and *HMP* were carried out from a batch of experimental measurements over the aforementioned standard.

[9]Among the following models implemented as an option within *scratch/error-model-test.cc*

hooked the parameter `m_rxTrace` at the moment immediately after the decision of whether a frame is correct or not (that is to say, at the end of the `ErrorModel::DoCorrupt` method).

Besides, there are two ways to access the information output that the `HiddenMarkovErrorModel` object yields: the aforementioned trace source (`m_rxTrace`), which can be hooked through the default `ns-3` traditional command, and a callback system (through the `m_rxCallback` variable). Anyway, you need to define a method which follows exactly these set of parameters, even respecting the order declaration (we will take the example used in `scratch/error-model-test.cc`):

```
void HmmRxTrace (Ptr<Packet> packet, Time timestamp, bool
    error, u_int16_t state);
```

Where:

- `Ptr<Packet> packet`. Copy of the received packet. It could be used for parsing the protocol headers headers (IEEE 802.11 MAC[10], IP, TCP/UDP).
- `Time timestamp`. We have carried out a performance analysis of the Hidden Markov Process model, thus it is indispensable to keep track of all the reception timing, if we want to study the evolution of the wireless channel during a transmission.
- `bool error`. This value returns "1" when a frame is received correctly
- `u_int16_t state`. This variable set the state belonging to the concrete received frame. Through these parameter we can check the overall percentage of correct/erroneous frames that each state actually yields. Besides, we are able to

Callback connection

Due to the inherent complexity that a traced attribute may bring about if we want to link to, we have carried out another alternative to connect the output system that the HMM supports. In this case, if we want to get hooked to the information that the method `HiddenMarkovErrorModel::DoCorrupt` returns, we only need to link our own method or function by typing the following lines:

1. First of all, we need to instance a `HiddenMarkovErrorModel` object.

   ```
   Ptr <HiddenMarkovErrorModel> errorModel = CreateObject <
       HiddenMarkovErrorModel> ();
   ```

2. Once defined the error model, we just need to link the callback with our proprietary function.

[10]The physical header is dropped before reaching this method.

```
errorModel->SetRxCallback(MakeCallback(&Experiment::
    HmmRxTrace, &exp_));
```

4 Applying the patch

In order to easily merge the new model with the legacy ns-3 code (remark that this patch does only work with the **ns-3.13** version), we have made a patch by means of the `diff` command, thus applying is as simple as just typing the following command:

patch -p1 < error-models.patch

Nevertheless, it is worth highlighting that the patch file **must** be pasted into the `ns-3-dev` folder.

References

[1] Y. Ephraim and N. Merhav, "Hidden Markov processes," *Information Theory, IEEE Transactions on*, vol. 48, no. 6, pp. 1518 –1569, jun 2002.

[2] "`ns-3` coding style." [Online]. Available: http://www.nsnam.org/developers/contributing-code/coding-style/

[3] "`ns-3`." [Online]. Available: http://www.nsnam.org/

[4] "Gnu coding standard." [Online]. Available: http://www.gnu.org/prep/standards/