# *Ns*-based Bluetooth LAP simulator[†]

Choonhwa Lee and Abdelsalam (Sumi) Helal

Computer and Information Science and Engineering Department
University of Florida, Gainesville, FL 32611-6120, USA
*{chl,helal }@cise.ufl.edu*
*http://www.harris.cise.ufl.edu*

***Abstract***

*We present a Bluetooth LAN Access Point (LAP) simulator that we have developed to study pervasive application behavior under Bluetooth local connectivity. Our goal is to explore the impact on IP applications caused by the underlying Bluetooth protocol. Our simulator implements detailed processing of upper layers of Bluetooth LAP stack, including PPP, RFCOMM, and L2CAP. For lower layers, it tries to capture major characteristics by performing macro-simulation. We present simulation results of a simple network configuration in Bluetooth LAP environments. Our simulator is based on the popular Network Simulator (ns) and its components.*

## 1. Introduction

*Ns* is a popular discrete event network simulator that offers a comprehensive set of simulation components. It allows researchers a great deal of flexibility in composing these components into specific networks. *Ns*' built-in modules include application, traffic, transport, queuing, routing, wired/wireless LAN, and other supporting modules. Using a script language, users can easily assemble these components to build their simulation networks and invoke various simulation scenarios. The *ns* simulator provides substantial support for simulation of TCP, routing protocols, multicast protocols, and wired/wireless network protocols [1].

*Ns*' simulation engine is written in C++ and uses OTcl as a frontend to configure simulation scenarios. Each component is implemented in C++, since C++ is fast and suitable for detailed protocol implementation such as packet header processing and algorithm running over large data set. More sophisticated protocol behavior can be implemented by extending base C++ components. *Ns* is still growing, although it already supports a large number of components for network simulation: *Scheduler, Timers, Node, Packet, Links, Error Model, Queues, Delays, wired/wireless LAN protocol, Agents (TCP, UDP, Multicast), Routing Protocols, and even Applications (FTP, telnet, and Web)*. The hierarchy of these C++ classes is mirrored by a hierarchy of classes in OTcl, which are control points for users. In other words, OTcl command is used to assemble core components, written in C++, to compose more complex object. Using OTcl command, users can set parameters for a simulation module, construct a node by linking protocol layers within it, and describe network topology. Moreover, OTcl is used to define simulation scenarios and control simulation runs. OTcl provides an excellent control and configuration interface to the users.

The use of two languages enables the users to reap the best of C++ and OTcl, i.e., performance and extensibility of C++ classes and composability and configurability via OTcl classes. In addition, *ns* has visualization tools such as network animator (*nam*) and graph generator (*xgraph*) to help the users analyze their simulation results.

### 1.1 Bluetooth LAP Simulator Scope

In this paper, we explore application behavior in connection with Bluetooth local connectivity. More specifically, the ultimate goal of our simulator is to investigate the end-to-end impact on IP applications requiring communication between a mobile Bluetooth node and an Internet node through a Bluetooth LAP gateway. The *ns* simulator is a perfect starting point for this purpose, since its problem domain subsumes ours and its extensible architecture guides us to add new protocol support to the ns' framework.

One of the target applications for our simulator is typical proximity computing scenarios, such as querying a Jini lookup service for a color printer service within proximity. To this end, a milestone has been set to provide a limited support of the Bluetooth protocol suite in the first (current) version of our simulator. The first phase is to simulate Bluetooth LAN access profile. It supports Bluetooth upper layers (PPP, RFCOMM, and L2CAP layer) as well as the macro-simulation of the HCI interface. For

[†] This research was fully funded by Motorola Corporation under grant number 45142551-12.

L2CAP, our simulator currently supports only connection-oriented channels. The connectionless data channel is not implemented by the current version. As part of HCI macro-simulation, ACL links for data is simulated but SCO link support is not. Lower layers such as LMP and Baseband are also out of the scope of our simulator. Other groups such as the NIST Networking Research group and IBM Research have developed simulation support for these lower layers. The IBM simulator is based on *ns*. The next version of our simulator is planned to support these missing Bluetooth protocol features, including node mobility and resultant impact on Bluetooth RF system.

The rest of the paper is organized as follows. Section 2 describes the basic architecture of our Bluetooth LAP simulator. Detailed implementation of each Bluetooth layer is presented in Section 3. Finally, section 4 presents preliminary performance results of simple Bluetooth LAP configurations obtained using our simulator.

## 2. Bluetooth Extensions to *ns*

### 2.1 Overall Architecture

We have used *ns* version *ns-2.1b6*. The node structure of our simulator, defined by *BluetoothNode,* a new OTcl class, is based on *ns' MobileNode*, even though we do not implement the details of Bluetooth mobility features in the first version. They are to be supported in the next version. Figure 1 shows the protocol stack for applications, incorporated into the *ns* framework. In our Bluetooth LAP simulator, some applications and TCP/IP components are reused without any change, while lower layers are replaced by *ns* components written for the Bluetooth LAP profile layers.

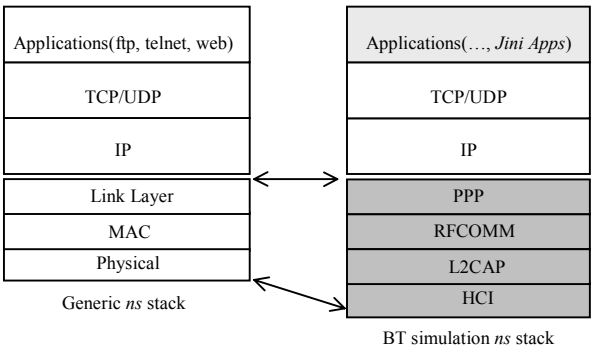


**Figure 1. TCP/IP stack in ns and Bluetooth LAN access profile stack**

Figure 2 shows the overall architecture of a Bluetooth node, which shows us how new modules are integrated into the *ns* framework. Mobile Bluetooth nodes and LAP nodes have the same node structure shown in Figure 2 with the only difference being the *mobility enable/disable* flag. Note that Bluetooth mobility features would be supported by the next release. Although a single Bluetooth stack is shown in the figure, a node can be configured to have multiple instances of the stack. Bluetooth LAP node would be the case where the LAP device needs to serve multiple Bluetooth nodes simultaneously. This multiple instance configuration is explained later in details.

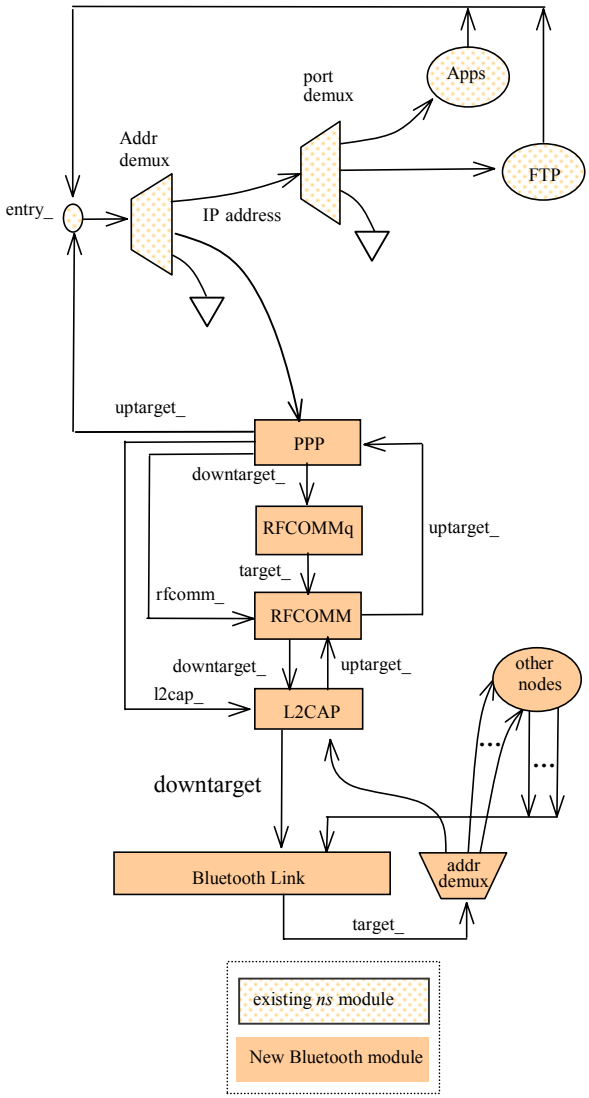


**Figure 2. Structure of a Simulator Node**

A packet sent by an agent such as application or FTP agent is handed to the address demultiplexer (*addr demux*) through the node's entry point (*entry_).*

Depending on the destination address of IP header, the packet is forwarded either to port demultiplexer (*port demux)* or to the Bluetooth network interface. If the packet is self-addressed, it is passed on to the destination agentd. Otherwise, this outgoing packet is sent to the PPP module. The packet ripples through PPP, RFCOMM, and L2CAP down to Bluetooth Link module. Packet propagation to the next module is handled through the *ns* standard inter-component pointers (*target_*, *downtarget_*, and *uptarget_*). In the case of unidirectional link, *target_* pointer is used and *downtarget_/uptarget_* pointers are used for bi-directional packet flows. Downward packets follow the *target_ or downtarget_* pointers. Note that Bluetooth Link module simulates Bluetooth RF link and, therefore, is shared by other Bluetooth devices in a piconet.

In case of incoming packets, the reverse path is taken. A packet sent by other devices arrives at L2CAP through *address demux* associated with the Bluetooth Link. It is routed to the PPP module by following *uptarget_* pointers. From there, the packet is handed over to the node's entry point. Finally, it is passed on to its destination agent through *addr demux* and *port demux*.

## 2.2 Simulator Input & Output Format

A typical *ns* input script performs several tasks as follows: First, it describes the simulated network topology, which includes node configurations, network interface configurations inside of a node, communication links, agents, and connection setups between agents. Next, simulation trace module needs to be set to generate simulation event logs for later analysis. Communication events are scheduled to happen at a specified simulation time. Finally, the simulation is started.

Our input script consists of Tcl procedures added by our Bluetooth extension as well as standard *ns* procedures. Among our additions, important procedures are explained in the following protocol module section.

The simulator produces trace information about events that have occurred during a simulation run. Each trace line represents a communication activity (for example, send, receive, or drop event) within a simulation component. We have extended the *ns* trace format to record Bluetooth events. The trace log is composed of a common part and a Bluetooth-specific part. The common part, which is fields (1) thorough (9) as shown below, is the information common to IP traffic. These fields are originally defined by *ns*. Bluetooth-specific parts are our addition to represent events related to Bluetooth traffic. A sample line may look as follows.

```
(1) (2)          (3) (4)      (5)(6) (7)                      (8)                    (9)
s   7.627856000 _3_ BLT  --- 5 tcp 1031 [0 0 0 0] ------- [4196352:0 0:0 32 0] [2 0] 0 0

(10)      (11)           (12)

[IP data][2 UIH 0 1022][1029 65 -co data-]
```

1) *An "r" indicates a packet receive event at a simulator node, whereas "s" indicates a send packet event. A "d" means a packet drop event.*
2) *Time when the event occured.*
3) *Node identifier.*
4) *Trace group: BLT represents an event related to Bluetooth traffic.*
5) *Packet identifier identifies a packet exchanged between a sender node and a receiver node.*
6) *Protocol layer identifier.*
7) *PDU size represents actual packet size.*
8) *IP information.*
9) *TCP information.*
10) *PPP information: protocol id plus either data or "packet type, identifier, and length" in case of LCP packet.*
11) *RFCOMM information: address, frame type, P/F bit, and length.*
12) *L2CAP information: length and channel id plus either -co data- or "code, identifier, and length" in case of signaling packet.*

## 3. Simulator Bluetooth Modules

In this section we give important details about how we implemented the new Bluetooth modules into *ns* extensions. The details may not be of interest to many readers, but are of great importance to researchers and developers currently working on building Bluetooth simulators.

### 3.1 Applications

One of the ultimate goals of this work is to study the behavior of proximity computing applications enabled by Bluetooth as the wireless local connectivity technology. The Jini service discovery scenario is a perfect candidate for this purpose, in that it involves inherently both local connectivity protocol and impromptu service discovery at the application level. When a mobile device moves into a local network with Bluetooth LAPs deployed, it discovers and connects to one of them. Over an IP connection provided by one of the LAPs, a mobile Bluetooth device can access a Jini federation by contacting its lookup server. If the desired service is found, it is downloaded over the LAP and is invoked by the mobile device.

We intend to see how these proximity applications perform over a Bluetooth link. We will take a sample of Jini application benchmark on a real network and re-apply it to our simulator. The *ns' TrafficGenerator* is used to feed the sampled traffic pattern to our simulator. The result of Bluetooth LAP simulation presented in section 4 will serve as the baseline of this application simulation. Taken together with the LAP simulation, the proximity application simulation will give us a better understanding of the characteristics or requirements of proximity computing enabled by Bluetooth.

### 3.2 PPP

The Point-to-Point Protocol (PPP) provides a standard method for transporting multi-protocol datagrams over point-to-point links. Namely, PPP specifies: 1) a method for encapsulating multi-protocol datagrams and 2) a Link Control Protocol (LCP) for establishing, configuring, and testing the data-link connection [6]. The simulator implements the core functionality: LCP connection establishment and encapsulation of IP packets. Namely, the state machine of LCP is implemented and encapsulated IP packets are sent when the state of LCP reaches *Opened* state. However, optional PPP PAP, CHAP, and IPCP protocols are not covered by our simulator. A PPP connection between two nodes is created statically by putting the following line in the simulation script.

```
$ns_ create-ppp-connection $node_(0)$node_(1)
```

The procedure *create-ppp-connection* creates a PPP interface consisting of PPP, RFCOMM, and L2CAP modules on both source and destination nodes (by calling *add-interface-ppp* procedure on each node). The *add-interface-ppp* procedure instantiates the Bluetooth stack shown in Figure 2. Then, each L2CAP entity of both ends is attached to the Bluetooth link that represents a connection between a master device and a slave device in the piconet (by calling *add-to-bluetoothlink* procedure). As a side effect of the last step, routing table is adjusted, so that traffic destined to the peer node can be directed to this instance of PPP interface.

### 3.3 RFCOMM

The RFCOMM protocol provides emulation of serial ports over the L2CAP protocol [8]. The RFCOMM protocol is defined by specifying a subset of the ETSI TS 07.10 standard, along with some Bluetooth-specific adaptations. The RFCOMM is a simple transport protocol, with additional provisions for emulating the 9 circuits of RS-232 (EIATIA-232-E) serial ports.

The RFCOMM module is implemented by subclassing C++ *Biconnector* class of *ns*. The reason for choosing *Biconnector* as the base class is that it has the *uptarget_* and *downtarget_* members. Note that RFCOMM layer needs pointers to upper and underlying layers. Between *ns* modules, packets are passed either by directly calling the target module's *recv()* method or by scheduling a receiving event at the target module. The scheduled event is handled by *handle*() and, in turn, the *recv*() method eventually. Therefore, implementing a module in the *ns* is all about how to write its *recv()* method. Following the standard style of the *ns* protocol layer modules, *recv()* method calls the *sendDown()* or *sendUp()* methods depending on the direction field in the common header. They are the embodiment of the RFCOMM layer.

The RFCOMM uses the basic option of TS 07.10, which defines a multiplexer that has a dedicated control channel: DLCI (Data Link Connection Identifier) 0. This channel is used for signaling purposes or conveying information between two multiplexers. When an RFCOMM entity receives a packet from its upper layer, it first checks if there is an active connection to the peer. If not, it creates a multiplexer session by following standard TS 07.10 signaling procedure. On successful connection, it

makes a new DLC connection to the peer by sending a signaling message (SABM frame) on that session (DLCI 0). While this connection setup process is underway, all data from PPP layer are queued at the RFCOMMq module. Once done, the RFCOMM module starts to process data packets from the RFCOMMq module.

## 3.4 L2CAP

L2CAP supports higher level protocol multiplexing, packet segmentation and re-assembly, and the conveying of quality of service information [10]. As in the case of the RFCOMM layer, the most appropriate base class for L2CAP module is *Biconnector*.

L2CAP provides a *reliable channel* to RFCOMM layer, which is provided by simple ARQ mechanism in the underlying Baseband layer. The Baseband layer always performs data integrity checks when requested and resends data until it has been successfully acknowledged or a timeout occurs (ARQ). Another important feature of L2CAP is protocol multiplexing. L2CAP must be able to distinguish among upper layer protocols such as RFCOMM and SDP, using PSM (Protocol/Service Multiplexer) field in *Connection Request* signaling message. Our simulator implements a simplified protocol state machine to keep track of the state of each channel. Currently, the simulator only supports connection-oriented channels. Connectionless channel support is not mandated by Bluetooth LAN access profile.

## 3.5 Bluetooth Lower Layers

Although our long-term goal is to analyze true Bluetooth protocol and associated application behaviors, our initial version commits to upper layers of Bluetooth LAP profile rather than simulating the whole stack, including LMP and Baseband layers. The low layer modules described below are serving as macro-simulation and temporary stopgaps until they are replaced by detailed implementations for a high-fidelity simulation in the next version.

We model a link (named *Bluetooth link* in this simulator) to capture several important features of the Baseband layer. These include bandwidth, delay, ACL packet types, error model, retransmission, and SAR (segmentation and re-assembly).

### 3.5.1 Bluetooth Link

Figure 3 shows the structure of Bluetooth link that provides an abstraction of the Bluetooth Baseband. *HCIlink_* component represents over-the-air link of Bluetooth with parameters such as bandwidth and delay. It flips packet direction from *DOWN* to *UP*. Positioned prior to the *HCIlink_* component, *errModel_* component introduces an error model to simulate packet loss by Bluetooth RF. When a packet is lost, some delay is inserted to capture a retransmission event by Baseband ARQ. The *ttl_* component discards packets when their ttl values drop to zero.

Whenever a PPP connection is created, each node is attached to the Bluetooth link, representing the fact that all devices within a piconet shall share the common Bluetooth radio link.

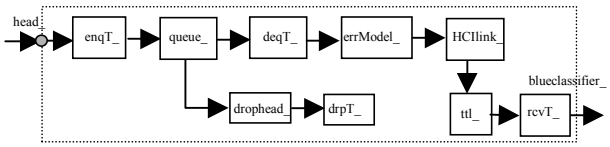


**Figure 3. Components of a Bluetooth Link**

### 3.5.2 Error Model and ARQ

The error model simulates packet corruption or loss at the Bluetooth link layer. In *ns*, several error models are available ranging from a simple packet error rate to more complex statistical or empirical models. Also, the unit of error can be specified in terms of packets, bits, or time. In a simulation input script, the following line declares an example error model to be used throughout the simulation.

```
$ns_ bluelink-err-config ErrorModel
0.05 pkt [new andomVariable/Uniform]
```

This sets an error model by which a 5 percent packet error is generated in a random fashion. If an error is detected by Baseband CRC checking, the packet is discarded and a simple automatic repeat request (ARQ) is invoked. With the ARQ scheme, packets are retransmitted until acknowledgement of a successful reception is returned by the destination (or the number of retransmission exceeds a predefined count). Notice that L2CAP layer relies on the underlying Baseband to provide reliable channels to RFCOMM layer. The ARQ scheme works on all types of packets used by the LAN access profile, since they all have CRC fields (AUX1 ACL packet is not used by L2CAP). In our simulator, ARQ retransmission is mimicked by the aforementioned error model. In other words, if an error occurs, the simulator inserts a delay equivalent to one resulting from the ARQ mechanism.

### 3.5.3 Segmentation and Re-assembly (SAR)

Six ACL packet types can be used in LAN access profile: *DM1, DH1, DM3, DH3, DM5, and DH5*. The ACL channel rate varies from 108.8 kbps up to 723.2

kbps, depending on the chosen packet type. The specification allows the negotiation of packet types to be used between a master and slaves. Moreover, this negotiated set of packet types can change during the communication by the request of either the master or the slaves. Besides the negotiation capability, devices can be configured to automatically adjust packet types depending on channel quality (*LMP_auto_rate* and *LMP_preferred_rate*).

The specific ACL packet types to use throughout the simulation run should be declared in the beginning of the simulator input script. Our simulator does not support the automatic adjustment of ACL packet type.

Before it is sent to a peer node, a data packet from the upper layer is segmented into blocks according to the packet type size used at Baseband. Recipient reassembles these blocks into the original packet, before it passes it up to its L2CAP layer. Our simulator implements this segmentation and reassembly feature.

## 4. Bluetooth LAP Simulation

The previous sections gave the details of our Bluetooth LAP simulator design and implementation. In this section, we present simulation results of a simple application scenario under a "typical" network configuration. In our sample scenario, a mobile node transfers an ftp data to its server node on the wired network through an associated Bluetooth LAP node.

There are two design schemes by which a LAP device and mobile devices determine their roles. Note that the LAP device can act as the master or the slaves. First, the LAP acts as the slave. When a mobile device comes in the range of the LAP, it starts an inquiry procedure and page procedure for a connection to the LAP. These procedures form a new piconet where there are only two members: the LAP and the mobile device. Consequently, the mobile device becomes a master device in the piconet. This approach has the penalty of time delay overhead taken by the LAP node (as a slave node) to switch among the different masters belonging to several unsynchronized piconets simultaneously.

In contrast, a LAP can play the master role and mobile devices would become slaves associated with it. In this approach, a new-coming device initiates an inquiry procedure and page procedure as in the first approach. On connection to the LAP, it requests master-slave role-reversal by *LMP_switch_req*. Then, the LAP becomes the master and the mobile device is added to the group of slaves associated with the LAP. This scheme enables the LAP to control slave's access to the link, since only the slave addressed in the master-to-slave slot is allowed to use the following slave-to-slave slot. But the number of slaves would be limited to seven in this approach.

In case a mobile device discovers multiple LAP devices to connect to in an area, it may need more information to select the most appropriate LAP; probably a least loaded LAP. The mobile device may get a hint on the availability of the LAPs from SDP's *ServiceAvailability* attribute.

Figure 4 provides the simplified view that represents the simulator configuration in case of a piconet where the master, i.e. the LAP node, communicates with 5 slaves. Recall that multiplexing is performed at the L2CAP layer on the LAP node.

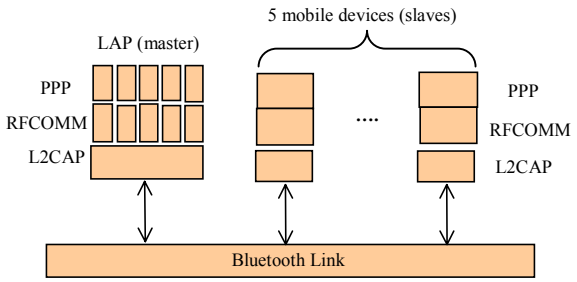


**Figure 4. A piconet in our simulator**

We ran a simple ftp scenario simulation on the one piconet configuration shown in Figure 4. The ftp scenario consists of an *FTP* source on a mobile Bluetooth device sending data at its full speed to its sink agent running on a wired node. The data is transferred to the wired node through the LAP node. The ftp throughput measured at the PPP layer of the mobile device is shown in Figure 5. Graph (a) is the throughput of a 1-to-1 master-slave configuration (a piconet of size 2), while graph (b) is the throughput of the five slave configuration shown in Figure 4 (a piconet of size 6). In case of (b), the measurement is taken on one of the five slaves (chosen at random), while they are sending data simultaneously.

Figure 5-a shows that throughput in the 1-to-1 master-slave case reaches a steady 640 Kbps almost instantaneously. This is consistent with our expectation. Figure 5-b, however, shows an average achieved throughput of 130 Kbps. The maximum throughput is achieved after an initial ramping period of about 15 seconds. This is due to initial contention of the five mobile devices at the LAP node.

Figure 6 shows the results of another set of ftp experiments where 3 different size files, 200 Kbyte, 800 Kbyte, and 3.2 Mbyte, are transferred in 2 to 8 node piconet configurations. As shown in Figure 6-a, ftp time increases significantly for a larger file, as the number of mobile nodes in the piconets increases. It is

well matched to our expectation that the Bluetooth over-the-air link would be a bottleneck in an overloaded piconet. Ftp throughput averaging 3 different size experiments are shown in Figure 6-b. The throughput reaches up to 640 kbps in 1-to-1 master-slave configuration, but it drops as multiple nodes contend to access the shared over-the-air channel.

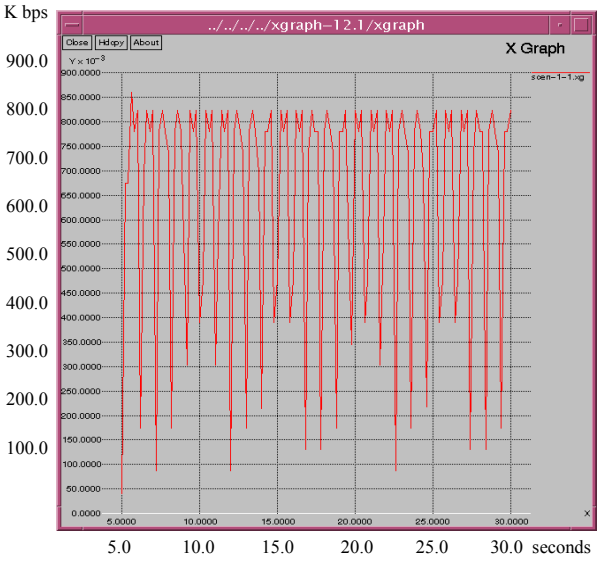


*Figure 5-a: 1 mobile device*

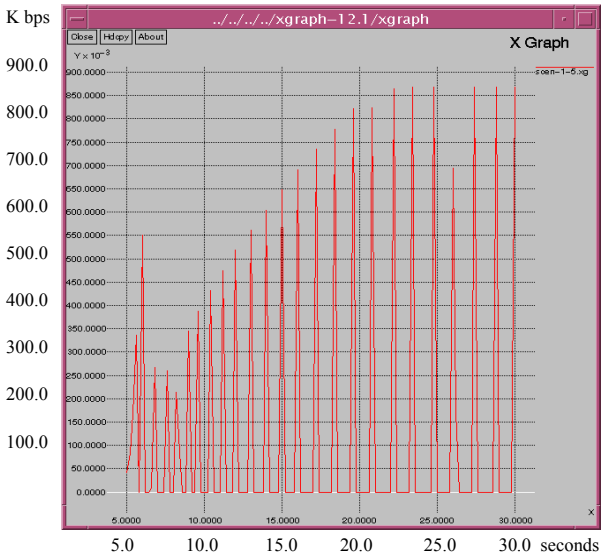


*Figure 5-b: 5 mobile devices*

**Figure 5. ftp throughput measured on one mobile device in 2 and 6 nodes piconets**

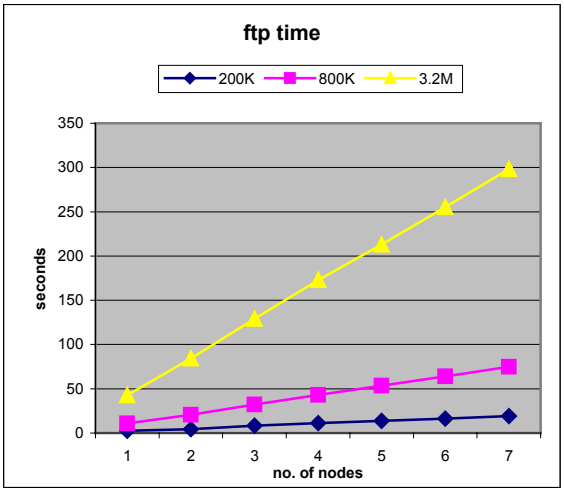


*Figure 6-a: time to ftp files of different sizes*

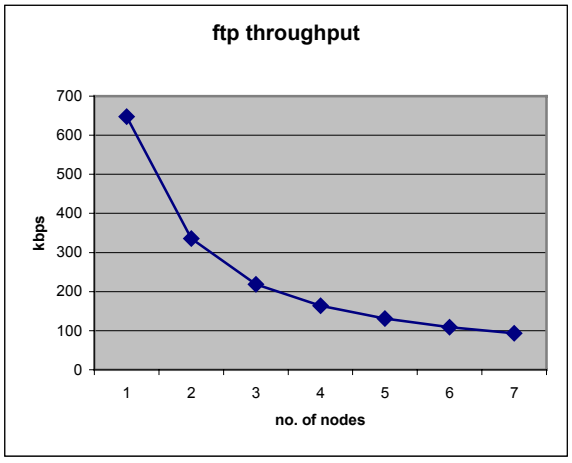


*Figure 6-b: average ftp throughput*

**Figure 6. FTP time and throughput measured on one mobile device in 2 to 8 node piconets**

## 5. Conclusion

We presented extensions that we implemented within *ns* to add Bluetooth protocol support. Our true motivation and goal in this work is to be able to study future proximity-based and pervasive computing applications in a Bluetooth local connectivity environment. Our immediate needs (and hence, highest priority) was to support a Bluetooth LAN access profile simulation within *ns*. This requirement defined the scope of this first version of the simulator. We have chosen *ns* as the base simulation platform because of the rich set of well-defined IP simulation components it already has. Also, its extensible and flexible architecture allowed us to easily add new components and compose simulation scenarios out of

them. Our simulator implements the upper layers of Bluetooth LAN access profile in detail, including PPP, RFCOMM, and L2CAP layers. For lower layers, it captures several important characteristics by doing macro-simulations. We also presented simulation results of simple network configurations in Bluetooth LAP environments.

To better understand the full impact of Bluetooth on future pervasive computing applications (e.g., performance, reaction to noise, and interferences in a piconet or scatternets), our simulator must be extended to capture the Bluetooth Baseband layer. This will be our next step. We may integrate existing Baseband layer simulation components implemented by other researchers in the filed (e.g., NIST Networking Research group, or the IBM-India Research Center group).

## 6. Acknowledgements

We would like to acknowledge several people who helped us in building this simulator. Joe Fernandez provided good insight during the early brainstorming of this project. Jaime Borras, Alif Khawam, Ruben Formoso, provided valuable comments. Yong Zhi helped in debugging and verifying the simulator.

## 7. References

[1] ns - Network Simulator. http://www.isi.edu/nsnam/ns

[2] ns Manual. http://www.isi.edu/nsnam/ns/ns-documentation.html

[3] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addition-Wesley Publishing Company, Inc., 1994

[4] An on-line Otcl tutorial, http://www.neosoft.com/neowebscript/nws2.3/commands/otcl/doc/tutorial.html

[5] Don Libes, A Debugger for Tcl Applications, Tcl/Tk Workshop, June1993, http://expect.nist.gov/tcl-debug/tcl-debug.tar.gz

[6] Simpson, W., The Point-to-Point Protocol (PPP), RFC 1661, Daydreamer, July 1994

[7] McGregor, G., The PPP Internet Protocol Control Protocol (IPCP), Merit, May 1992

[8] RFCOMM with TS 07.10, version 1.0 B, November 1999

[9] TS 07.10 (TS 101 369) Digital cellular telecommunications system (Phase 2+); Terminal Equipment to Mobile Station (TE-MS) multiplexer protocol (GSM 07.10 Version 6.3.0 Release 1997)

[10] Logical Link Control and Adaptation Protocol Specification, version 1.0 B, November 1999

[11] Host Controller Interface Functional Specification, version 1.0 B, November 1999