Advances in Pervasive Computing and Networking, Springer, New York, NY, 2004, pp. 249-267

Chapter 1

SENSE: A WIRELESS SENSOR NETWORK SIMULATOR

Gilbert Chen, Joel Branch, Michael J. Pflug, Lijuan Zhu, and Boleslaw K. Szymanski
Department of Computer Science,
Rensselaer Polytechnic Institute

Abstract

A new network simulator, called SENSE, has been developed for simulating wireless sensor networks. The primary design goal is to address such factors as extensibility, reusability, and scalability, and to take into account the needs of different users. The recent progresses in component-based simulation, namely the component-port model and the simulation component classification, provided a sound theoretical foundation for the simulator. Practical issues, such as efficient memory usage, sensor network specific models, were also considered. Consequently, SENSE becomes an ease-of-use and efficient simulator for sensor network research.

Keywords: Wireless Sensor Networks, Network Simulation, Component-Based Simulation

Introduction

The emergence of wireless sensor networks created many open issues in network design [**?**]. The three main traditional techniques for analyzing the performance of wired and wireless networks were analytical methods, computer simulation, and physical measurement. However, many constraints imposed on sensor networks, such as energy limitation, decentralized collaboration, and fault tolerance necessitate the use of complex algorithms for sensor networks that usually defy analytical methods. Furthermore, few sensor networks have come into existence, for there are still many unresolved research, design and implementation problems, so measurements are virtually impossible. It appears that

simulation is currently the primary feasible approach to the quantitative analysis of sensor networks.

ns2 (http://www.isi.edu/nsnam/ns/), perhaps the most widely used network simulator for research, has been extended to include some basic facilities to simulate sensor networks. However, one of the problems of ns2 is its object-oriented design that introduces much unnecessary interdependence between modules. Such interdependence sometimes makes the addition of new protocol models extremely difficult, which can only be mastered by those who have intimate familiarity with the simulator. The difficulties in extension are not a major problem for simulators targeted at traditional networks, for there the set of popular protocols is relatively small. For example, Ethernet is widely used for wired LAN, IEEE 802.11 for wireless LAN, TCP for reliable transmission over unreliable channels, etc. For sensor networks, however, the situation is quite different. There are no such dominant protocols or algorithms and there will unlikely be any soon. A sensor network is often tailored to a particular application with specific features, so it is unlikely that a single algorithm can always be the optimal one under various circumstances.

Many other publicly available network simulators, such as J-Sim [**?**], SSFNet (http://www.ssfnet.org), Glomosim [**?**] and its commercial descendant Qualnet, attempted to address problems that were left unsolved by ns2. Among them, J-Sim developers realized the drawback of object-oriented design and tried to attack this problem by inventing a component-oriented architecture. However, they chose Java as the simulation language, inevitably sacrificing the efficiency of simulation. SSFNet and Glomosim focus on parallel simulation, with the latter tailored specifically to wireless networks. They do not appear superior to ns2 in the respects of design and extensibility.

SENSE (SEnsor Network Simulator and Emulator) aims to be an efficient and powerful sensor network simulator that is also easy to use. We identify three most critical factors in its design as *extensibility*, *reusability*, and *scalability*. We distinguish also three types of users as *high-level users*, *network builders*, and *component designers*. In the next section, we explain what each factor implies and how SENSE meets the different needs of all users. In the sections that follow, we present in details the design decisions and implementation that are centered around these design factors and that take full consideration of needs of all three types of users. Finally, we will compare the performance of SENSE with that of NS using the flooding simulation as a benchmark.

1. Design Philosophy

1.1 Extensibility, Reusability and Scalability

The enabling force behind the fully extensible network simulation architecture in SENSE is the recent progress in component-based simulation [**?**]. A *component-port model* frees simulation models from interdependence usually found in an object-oriented architecture, and a *simulation component classification* naturally solves the problem of handling simulated time. The component-port model makes simulation models extensible: a new component can replace an old one if they have compatible interfaces, and inheritance is not required. The simulation component classification makes simulation engines extensible: advanced users have an option of developing new simulation engines that meet their special needs.

The removal of interdependence between models also promotes reusability. A component developed for one simulation can be used in another if it satisfies the latter's requirements on the interface and semantics. In SENSE, another form of reusability is made possible by the extensive use of C++ template. A SENSE component is usually declared as a template class so that it can handle different types of data, depending on the type parameters used to instantiate the component.

Unlike many other parallel network simulators, especially SSFNet (http://www.ssfnet.org) and Glomosim [**?**], parallelization will be provided as an option to the users of SENSE. This decision was based on our belief that completely automated parallelization of sequential discrete event models, however tempting it may seem, is impossible. Even if it were possible, it would be doomed to be inefficient as compared to hand-tuned parallel code. Therefore, parallelizable models must require much more effort and time than sequential models, while many users are not interested in parallel simulation at all. In SENSE, a parallel simulation engine will be capable of executing an assemblage of compatible components. If a user is content with the default sequential simulation engine, then every component in the model repository can be reused.

1.2 High-Level Users, Network Builders and Components Designers

High-level users solely rely on the model repository and network template library from where they can retrieve various network models and configurations to construct a sensor network simulation. For them, the process of building a simulation merely consists of selecting appropriate models and templates and perhaps changing some parameters. Such

users may not need any programming skills. Extensibility and reusability are not their concerns, but they may want the simulations to be scalable.

The network builders are not satisfied with the available network templates, but they still rely on the model repository to obtain network models. They may need to create new network topologies and traffic patterns. These users may not have immediate or knowledge of popular programming languages, such as c/c++, Java. Extensibility is not an issue for them, since they are not interested in modifying the existing models. However, models must be reusable so that they can be plugged into many simulations.

The component designer often intend to modify available models or even build new ones from scratch. For example, they can develop a proprietary MAC layer protocol which replaces the standard one. Their main concern is the extensibility; how easily existing models can be extended or replaced determines the willingness of these users to use the simulator. Reusability may or may not be an issue, depending on whether the new model is intended to be used in other simulations. The biggest challenge of the design for these users is to make the modeling process smoother, faster, and more reliable. The simulator should provide facilities to speed up checking, debugging, and verification of the models; there must be visualization tools to help identify any problems quickly; there must be standards that these users can follow in order for the models to be more accessible by others.

2. Component-Based Design

SENSE is built on top of COST [**?**], a general purpose discrete event simulator. The design of COST was largely influenced by the new understandings of both component-based software architecture and component-based simulation. Specifically, a component-port model was proposed to allow complex software systems to be built as a composition of components. Later, it was extended to the simulation domain where components are categorized into different types, based on how simulated time is dealt with.

2.1 Component-Port Model

In the component-port model, a component communicates with others only via *inports* and *outports*. An inport implements a certain functionality, so it is similar to a function. In contrast, an outport serves as an abstraction of a function pointer: it defines what a functionality it expects of others.

The fundamental difference between an object and a component in the component-port model is that the interactions of a component with others can be fully captured by the interface, while this is not the case for an object. For instance, an object is allowed to call member functions of any other object if it keeps a pointer or a reference to that object. Such communication, however, is not reflected in the interface or declaration of the object, and becomes manifest only when the implementation code is being examined. The resulting problem is that any function call to external objects will introduce implicit interdependence between objects, preventing the object from being reusable.

The existence of outports distinguishes components from objects. Outports impose constraints on the dynamic runtime interaction between components. The important consequence of their existence is that the development of a component can now be completely separated from the application context in which the component will be used, leading to truly reusable components. Besides, components become more extensible, because there are fewer constraints on a component that provides the desired functionality. For instance, in an object-oriented environment, if an object A is to be replaced by another object B, object B has to be derived from A. In the component-port model, this constraint is no longer necessary. Any component providing the satisfied functionality can be used, regardless of its component type.

2.1.1 Implementing Components. The subsequent task for us is to implement the component-port model with C++, a programming language that is usually regarded as object-oriented. Fortunately, we found template-based techniques can be utilized to archive this goal, although there are certain limitations due to the object-oriented features of the language.

First, we declare an *mfunctor* class that represents function objects for member functions of class *TypeII*. *TypeII* is the main component class, and we will explain why it is so called later in this section. The *mfunctor* class overrides the *operator()* function, so it can be called the same way as a normal function. Since it keeps a pointer to the component, it can be used to call the member function of any object derived from *TypeII*, if initialized correctly.

```
template <class T>
class mfunctor
{
 public:
  typedef void (TypeII::*funct_t)(T&);
  mfunctor(TypeII* _obj, funct_t _f)
```

```
    :obj(_obj),f(_f) {}
  void operator() (T& t) { (obj->*f)(t); }
 private:
  TypeII* obj;
  funct_t f;
};
```

The *inport* class is just a wrapper class that extends *mfunctor* so that the latter can be more conveniently initialized and invoked. To initialize an inport, a pointer to the component and a member function must be provided.

```
template <class T>
class inport
{
 public:
  void Setup(TypeII * c, mfunctor<T>::funct_t f)
  {
    functor = new mfunctor<T>(c,f);
  }
  void Write(T& t) { (*functor)(t); }
 private:
  mfunctor<T> * functor;
};
```

The *outport* class maintains a pointer to the inport to which it is connected. The *Connect()* function can be called to initialize this pointer. When the *Write()* function of *outport* is called, the *Write()* function of *inport* will be called, which in turn will invoke the member function of the component that was used to initialize the inport.

```
template <class T>
class outport
{
 public:
  void Connect(inport<T>&_in) { in=&_in;}
  void Write(T& t) { in->Write(t); }
 private:
  inport<T>* in;
};
```

One drawback of implementing components as stated above is that the inter-component communication may become quite costly, as the C++

compiler cannot completely optimize away the overhead of these function calls. However, it is possible to develop an optimization technique which can eliminate such communication overhead by merging components together so that the function to be called can be directly embedded into the code that makes the call, much the same as how inline functions work.

Another problem with the above implementation is that member functions are limited to take only one argument, as in standard C++ template classes with different numbers of template parameters cannot be given the same name. This problem can be solved by the use of wrapper classes around several arguments to make them appear as a single argument.

2.1.2 Components for Sensor Network Simulation. The component-port model gives the users a great deal of freedom in configuring sensor nodes. Figure 1.1 shows the internals of a typical sensor node. The sensor node is a composite component. It consists of a number of smaller primitive components, each implementing a certain functionality. Normally a sensor node has some layered network protocol components, a power component and a battery component both of which are related to power management, and others such as mobility and sensor. The inports and outports of the sensor node component are directly connected to the corresponding inports and outports of internal components.

This structure, however, is by no means the only one that users must strictly follow when they are building their own nodes. The user can freely remove or add a component, as demanded by the particular goal of the simulation. For instance, the network protocol stack can be either simplified by removing the net component, or tuned up by adding a new transport layer without affecting any other components. A queue component can be easily added between the network layer and the mac layer to prevent packets from being dropped when the mac layer is busy transmitting other packets.

In theory many programming languages can be used to configure sensor nodes into a network. Configuration is as simple as setting the parameters of all components and then interconnecting their inports and outports. In this phase, components do not communicate with each other, so any object-oriented language is sufficient to perform the task. Currently, C++ is chosen to be the only configuration language, since it is also the implementation language for components. The simplicity of the configuration does allow such languages as TCL or XML to be used.

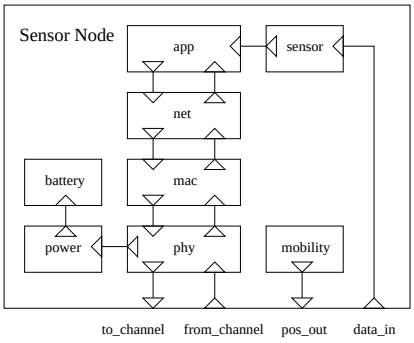

Figure 1.1. The internal structure of a typical sensor node

In addition, it is quite natural to develop a simple scripting language specifically for the network configuration phase.

2.2 Simulation Component Classification

The component-port model clarifies the role of components in the development of general software systems. It still remains unknown, however, how the component-port model can be applied to simulation. The answer lies in a simulation component classification that naturally extends the component-port model to the simulation domain [**?**].

According to this classification, based on the way how simulated time is handled, simulation components are grouped into *time-independent*, *time-aware* and *autonomous* classes, also named Type I, Type II and Type III classes, respectively.

A Type I component does not have the notion of simulated time. It is passive, as it never generates events without first having received an event. A Type I component, when processing an event received from other components, may generate new events that are required to have the same timestamp as the incoming event that triggered it. Yet, the component itself is unaware of the time semantics. Neither does it know whether it is running as a part of a simulation program or a part of a non-simulation program. For this reason, a time-independent component is said to be time-unaware.

In contrast, Type II components are time-aware components. They cannot advance the simulated time themselves, but they can make a time advance request via a special object called a *timer*. Timers provide a mechanism for Type II components to generate events whose timestamp is greater than the current simulated time. To schedule such a future event, a timer is set with a time increment representing the difference

between the current simulated time and the timestamp of the future event. As soon as the specified simulated time increment elapses, the component where the timer resides will be notified and then forced to process the activated event.

Type III components are named autonomous components because they maintain their own simulation clock themselves. A *clock* indicates the simulated time throughout the simulation. A sequential simulation is a Type III component by itself, which does not communicate with other Type III components. In parallel simulation, there are usually several Type III components, each mapping to a process or thread. These Type III components have to be synchronized by certain algorithms so that they can interact with each other correctly by exchanging events.

The simulation component classification leads to a hierarchical modeling process in SENSE. Because of the composability of components, a number of components can be combined into a single component. However, this kind of composition does not change the component type. If every individual component is of Type I, so will the composite component. If at least one of them is of Type II, then the composite component will also be of Type II. A *simulation engine* changes the type of the component. A simulation has to be a Type III component, so usually building a simulation involves deployment of one or several simulation engines.

This hierarchical modeling process distinguishes SENSE from many other parallel network simulators. There, the simulation engines are often built-in, and therefore users are forced to use the simulation engines provided by the simulator designers. Advanced SENSE users are given the option of building their own simulation engines, as the particular application they are investigating may call for a specific simulation algorithm (as of the time of this writing the parallelization of the simulator is still in progress).

3. Packet Management

A network simulation is composed of two types of entities: one are the static components that simulate various network devices and the other the dynamic packets that are created, transmitted, and received by components. The previous sections all dealt with only the simulation models, and we still need a good packet management scheme to effectively manipulate the packets. It turns out that this is not a trivial problem.

Our main consideration for the packet management is that it must be memory-efficient. Memory has become the most serious bottleneck that

prevents large simulation programs from running on computers equipped with limited memory. Because of the extremely slow disk access speed, programs that rely on virtual memory are often an order of magnitude slower than those that can fit into the physical memory. For this reason, we decided to design a packet management scheme that consumes as little memory as possible.

This consideration makes the packet management scheme in ns2 unsuitable. In an ns2 simulation, every packet, no matter which protocol layer it belongs to, has to occupy the same amount of memory. It works well when protocol layers (other than the top one) do not create new packets, for instance, when each protocol simply appends its header to the packet and then forwards it to the lower layer. This is often not the case, however. A lower layer protocol may break a large packet into many smaller ones, as in fragmentation; it may also create new control packets, not including the original packet from the higher layer, as in handshake. In these cases, a considerable amount of memory would be wasted if we treated all packets as if they were of the same size.

Therefore, we came up with a layered packet structure, as shown in Figure 1.2. Each layer maintains its own packets, which usually consist of a header (denoted by H) and a payload field (denoted by P). The payload field contains either a pointer to, or a copy of, the packet at the intermediate upper layer. If the size of the upper layer packet is much larger than the size of a pointer, then a pointer instead of the packet itself can be kept, represented by dotted arrows; otherwise an actual copy of the packet, represented by solid arrows, will be more convenient.

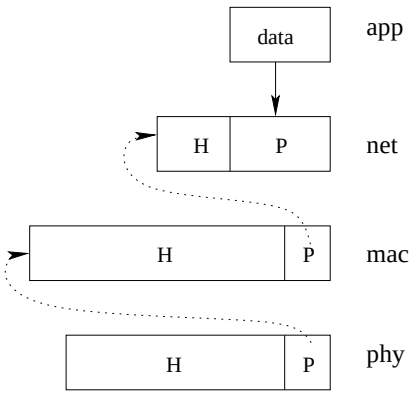

Figure 1.2. The Layered Packet Structure

Another decision we made regarding the packet management is that a packet sent by one node will be shared by all receiving nodes. This

is possible because it is usually meaningless to 'modify' the receiving packet. Wireless nodes always share the communication medium with neighbors, so it is expected that one packet will often be received by many nodes. Consequently, the amount of memory saved by this approach will be considerable.

A standard programming technique, *reference counting*, is adopted to keep track of packets. When a node receives a packet, it must increment the reference count of the packet to indicate that it now partly owns the packet. When a packet is to be released, its reference count will be decremented. Only when the reference count goes to zero can the packet be actually deleted.

However, such a packet structure results in an inevitable problem. Assume a scenario in which a certain layer asks the physical layer to transmit a packet by pointer. The physical layer may successfully transmit the packet out, in which case the pointer will be forwarded to other node. However, the problem arises when the transmission fails, for instance, if there are no other nodes within the transmission range. The packet has to be destroyed by the physical layer.

This implies that the lower layer may need to be responsible for releasing the pointer to the packet sent from any higher layer, and this problem is not limited to the physical layer, since other layers may attempt to drop packets under special circumstances. In general, no reliable transmission can be guaranteed.

On the other hand, if the payload field contains not the pointer to, but a copy of the packet from the upper layer, then no operation is needed when the packet is to be dropped. For any intermediate layer, packets from the higher layer could be in the form of either pointers or plain structures. It seems that we would have to implement two components for each layer, one accepting pointers and the other copies.

Fortunately, this problem can be elegantly solved by a C++ template technique referred to as *trait*. According to Bjarne Stroustrup, a trait is "a small policy object typically used to describe aspects of a type" (http://www.research.att.com/ bs/glossary.html). In SENSE, a special packet trait class is declared which can tell if a certain template parameter is a packet structure or a packet pointer.

The declaration of this packet trait class is shown below. Basically it means that for general packets, nothing needs to be done with regard to packet deallocation.

```
template <class T>
class packet_trait
{
 public:
```

```
  static void free(const T&) {};
};
```

The *smart_packet_t* class is the main SENSE packet class defined for layers other than the top one. It consists of a header and a payload field, as well as a reference count.

```
template <class H, class P>
class smart_packet_t
{
 public:
  ...
  inline void free();
  H hdr;
  P pld;
 private:
  int refcount;
};
```

In the *free()* function of the *smart_packet_t* class, it first calls the *free()* function of the payload via the *packet_trait* class. It then decrements the reference count, and if the reference count is zero, both the header and itself will be freed.

```
template <class H, class P>
void smart_packet_t<H,P>::free()
{
  packet_trait<P>::free(pld);
  refcount--;
  if(refcount==0)
  {
    packet_trait<H>::free(hdr);
    delete this;
  }
}
```

Below is the partial specialization of *packet_trait* for pointers to *smart_packet_t*. As a result, in the *free()* function given above, if the payload contains a pointer to a smart packet, the smart packet will be freed; for all other cases nothing happens. If users are to define their own packet types and keep track of them by pointers, they should specialize the *packet_trait* class in a similar way.

```
template <class H, class P>
```

```
class packet_trait< smart_packet_t<H,P>* >
{
 public:
  typedef smart_packet_t<H,P> nonpointer_t;
  static void free(nonpointer_t* const &p)
  {
    if(p!=NULL) p->free();
  }
};
```

4. Component Repository

As the core design of SENSE has been finalized, we built an extensive set of components ranging from application layer to physical layer, as well as energy and mobility models that are specifically targeted at sensor networks.

4.1 IEEE 802.11

The IEEE 802.11 component in SENSE implemented the distributed coordination function (DCF) described in the IEEE 802.11 standard. To transmit a data packet, this MAC component first checks the size of the data packet. If the size is smaller than a predefined threshold given by a parameter named *RTSThreshold*, or if the data packet is to be broadcast, the data packet will be transmitted directly, with a proper header added. If the size is greater than *RTSThreshold*, an RTS/CTS exchange mechanism will be invoked prior to the actual data transmission, in order to reserve the medium for a period of time that is just sufficient for the entire transmission. A unicast data packet must be accompanied by an acknowledgment, but not a broadcast data packet. A transmission is deemed successful only if the acknowledgment packet has been correctly received. Each failed transmission will double the content window until it reaches the preset maximum value.

The IEEE 802.11 implementation in SENSE has the same detail level as that of ns2 (http://www.isi.edu/nsnam/ns/). However, the source code in SENSE is twice as short as that in ns2, which can be attributed to the simplicity and effectiveness of the SENSE API. For example, timers are implemented as a template class that takes the type of event as a parameter. Defining a timer in SENSE is as simple as writing a statement to instantiate the timer. On the contrary, in ns2 each timer instance needs a unique implementation of a class derived from the base timer class, which greatly degrades the efficiency and readability.

4.2 AODV

Ad-hoc on demand distance vector routing (AODV) has been well-received as a routing protocol for wireless networks. AODV's route discovery consists of setting up a forward and reverse data transmission path between two mobile nodes. After route discovery is complete, each node belonging to the established path maintains a routing table via sequenced requests and response messages. A table entry primarily consists of two IDs: one denoting the destination node and the other denoting the next-hop node along the path to the destination. The sequence numbers included in the request/response packets ensure that these routes are loop-free. Other table entry information is used to maintain route freshness, so that outdated route entries may be properly replaced. AODV's route maintenance also provides facilities for replacing damaged routes (e.g., those with broken links). Each node maintains only partial (local) route information, so full path information is never transmitted between nodes. A seminal document [**?**] provides more details about AODV.

The AODV implementation in SENSE is based on the most current AODV internet draft [**?**]. We have implemented the operative components essential to AODV's basic operation. This set includes all steps required to actually build routes. However, selected route maintenance functions have not been included in the current simulation. For example, provisions noted in section 6.8 of [**?**] for handling of unidirectional links have not been implemented. This is primarily because we only assume bi-directional links in our simulation. We have not yet included full facilities for maintaining local connectivity, processing route error packets, or implementing local repair functions. All these are expected to be completed in the near future.

4.3 DSR

Dynamic Source Routing (DSR) [**?**] is another widely used on-demand routing protocol for wireless networks. Similar to AODV, DSR provides a mechanism of route discovery if the route from the source to the destination is unknown. But unlike AODV, after the route has been discovered, the entire route is included in the packet header, and intermediate nodes will determine the next hop by looking at the routing information contained in the packet.

An initial version of the DSR Component for SENSE has been completed which makes certain restrictive assumptions within DSR specifications. Specifically, all nodes are assumed to be bi-directional, without support for promiscuous communications, and running in a homoge-

neous link layer environment. Moreover, we assume that all communication links, once established, are not subject to damages, and hence error handling and route recovery are not necessary. Our testing environment currently consists of DSR running on top of the 802.11 link level component, for which all of these assumptions are valid.

As DSR matures, and new upper-level and lower-level networking components are created, a number of the current limitations will be removed. An Immediate plan is to include route error packets so that the network can recover from faulty nodes or communication obstacles. Other plans include support for the promiscuous mode operation, the optional DSR flow state extension, uni-directional links, and a data link layer which does not provide acknowledgment information for unicast packets.

4.4 Battery Models

Two battery components have been implemented in SENSE. In the *SimpleBattery* component, the discharge rate is always proportional to the power drawn from the battery, and is not dependent on the current. Its capacity is a constant defined by the simulation parameter. Let E' be the previous remaining energy and P the power consumed in the time unit, the energy remaining after a consumption period of t can be expressed as:

$$E = E' - Pt \tag{1.1}$$

In the more complex *RealBattery* component, the discharge rate becomes dependent on the current: larger current usually renders the battery discharge quicker, thus resulting in less actual capacity at the end of the usage period than the smaller current would do [**?**]. A discharge rate dependence parameter, k, determines how the value of the current affects the discharge rate. More specifically, Equation 1.1 becomes:

$$E = \frac{E'}{1 + kI} - Pt \tag{1.2}$$

The *RealBattery* component also models relaxation [**?**], which refers to the phenomenon that a battery may gradually recover some of its lost capacity if the discharge current undergoes a sudden drop to become very small. For simplicity, we assume that relaxation only occurs if the current first sustains for a fast discharge period of at least T_R with a current larger than I_R, and then suddenly drops from above I_R to 0. Let λ be the recovery rate, g the growth ratio that can be eventually

reached, then during the relaxation period the capacity is governed by the following equation:

$$E = gE'(1 - e^{-\lambda t}) \tag{1.3}$$

A restriction is imposed to ensure that the capacity after the relaxation period would not exceed the capacity right before the fast discharge period.

In this component, another parameter is provided to turn the relaxation off. If there is not relaxation, and if k, the discharge rate dependence parameter, is zero, the component regresses to the *SimpleBattery* component.

4.5 Power Model

In SENSE, the power component is responsible for power management. Currently, a *SimplePower* component has been implemented, which can operate on any of 5 modes: TRANSMIT, RECEIVE, IDLE, SLEEP, and OFF. 4 parameters specify the energy consumption rate under each of the first 4 modes, while in the OFF mode there is no energy consumption.

The power component accepts control from networking components. In response to the control signal, it can switch from one mode to another. Depending on its operating mode it also draws corresponding current from the battery.

5. Performance Comparison

To test the performance of SENSE in terms of execution speed and memory efficiency, we carried out a set of experiments that compared SENSE with ns2.

All simulations were conducted using a Dell Latitude D600 with an Intel 1.6 Ghz Pentium-M processor and 512MB 266MHz DDR SDRAM. The flooding simulation was used as the benchmark for comparison. The flooding implementation in the ns2 distribution was modified to minimize the memory usage. In the original implementation, each node maintained a hash table that stored every packet that has been received. After the modification was applied, each node would only store the latest sequence number for each source. Any packet that comes from a source with a sequence number smaller than the latest sequence number known for this source is deemed as having been received before. This modification greatly reduced memory consumption, and is in accordance with the flooding implementation in SENSE.

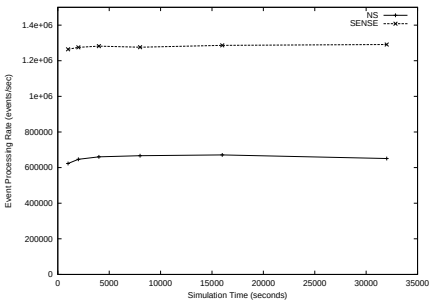

Figure 1.3. Event Processing Rate of NS and SENSE

For the comparison, TCL and C++ scripts were written to randomly generate traffic and topology files, and both simulators were modified to read from the same input files. All nodes are running the IEEE 802.11 protocol, but using only the broadcast functionality due to the nature of flooding. Simulations were conducted to compare the two simulators execution times and memory usage under various conditions.

All NS-2 simulations were conducted using NS-2 version 2.26. A few changes were made to the flooding TCL script that comes with the ns2 distribution to disable the simulator from producing the trace file. The heap scheduler was used in both, because it is less sensitive to different time increment distributions. Unnecessary headers were also removed to minimize the size of each packet.

We compared the execution speeds of both simulators. We created a wireless sensor network containing 60 nodes, with the same random placement and a 1000m by 1000m terrain. 12 sources were randomly chosen to send packets with a length of 1000 bytes, at fixed intervals of 10 seconds. Figure 1.3 shows that SENSE is consistently twice as fast as ns2. In both simulators the number of events were roughly the same.

The dramatic performance difference between ns2 and SENSE can be largely attributed to the ways they allocate and release packets. In ns2, when a packet is being broadcast, every neighboring node will receive a copy, so the number of packet allocations is equal to the number of received packets. In SENSE, a packet is always shared by all receivers, so the number of packet allocations is equal to the number of sent packets. In a dense wireless network, a node can usually communicate with dozens of neighbors. Consequently the number of received packets is far greater than the number of sent packets. Figure 1.4 confirms this explanation.

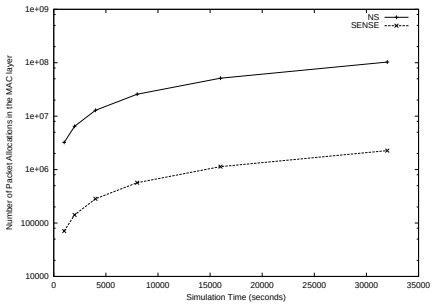

Figure 1.4. Frequency of Packet Allocation in ns2 and SENSE

6. Related Work

As stated in the introduction section, the development of SENSE was largely motivated by the realization of the fundamental drawback in the object-oriented designed of ns2 (http://www.isi.edu/nsnam/ns/). Compared with ns2, SENSE is not only more efficient, as shown by last section, but also more advanced in the architecture design since SENSE greatly promotes the reusability and composability of network models.

J-Sim [**?**] is also claimed to be a wireless network simulation with a component- oriented architecture. However, the inter-communication efficiency was not taken as a principal design factor, and as a result the overhead is larger than in the current version of SENSE. More specifically, in every J-Sim component, a *process()* function handles incoming events for all ports, which involves dynamic dispatch of events based on the ports that they come from. However, this mechanism incurs unnecessary run-time overhead, since communication between components can be largely deduced statically from their connections.

Several other simulators devoted to wireless sensor networks have been in progress. Among them, TOSSIM [**?**] and Emstar [**?**] are similar to each other in that both are a combination of a simulator and an emulator that can facilitate the development and deployment of sensor nodes. SensorSim [**?**] is basically a sensor network extension of ns2, while SensorSimII [**?**] has been rewritten in Java but still inherited the object-oriented design. SENS [**?**], being developed at UIUC, is another object-oriented sensor network simulator.

7. Conclusion and Future Work

The most significant feature of SENSE is its balanced consideration of modeling methodology and simulation efficiency. In designing SENSE,

we attempt to convey a belief that it is possible to build a very user-friendly simulator that is also very fast. Unlike object-oriented network simulators, SENSE is based on a novel component-oriented simulation methodology that promotes extensibility and reusability to the maximum degree. At the same time, the simulation efficiency and the issue of scalability are not overlooked. We observed that memory is the major factor that limits the size of simulation that can be actually performed, and that many other simulators contain too much overhead with respect to memory usage. The simulator is therefore memory-efficient, fast, extensible, and reusable.

SENSE is still in its active development phase. Although the core of the simulator has been gradually stabilized, it still lacks a comprehensive set of models and a wide variety of configuration templates for wireless sensor networks. Besides, a visualization tool is desirable which can quickly track down what goes wrong during the simulation. Without such a tool, the output of the simulation is hard to interpret. Visualization can also facilitate the configuration phase by allowing networks to be constructed graphically.

The problem of inefficient inter-component communication can be completely solved very soon. We have designed a component extension to the C++ language. The new language extension introduces only four keywords and four syntactic rules, with simple semantics that are easy to understand. It will not only improve the simulation speed, but also free SENSE users from the constraint that limits the number and granularity of components that can be used when efficiency is the main concern, since the inter-component communication overhead will be entirely eliminated with this new language extension.

References

Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cyirci, E. (2002). Wireless sensor networks: A survey. *Computer Networks*, 38(4):393–422.

Chen, Gilbert and Szymanski, Boleslaw K. (2002). COST: Component-oriented simulation toolkit. In *Proceedings of the 2002 Winter Simulation Conference*.

Girod, L., Elson, J., Cerpa, A., Stathopoulos, T., Ramanathan, N., and Estrin, D. (2004). Emstar: a software environment for developing and deploying wireless sensor networks. In *the Proceedings of USENIX General Track 2004*.

Hou, Jennifer, ying Tyan, Hung, et al. J-sim. http://www.j-sim.org/.

Johnson, D., Maltz, D., and Broch, J. (2001). *Ad Hoc Networking*, chapter DSR The Dynamic Source Routing Protocol for Multihop Wireless Ad Hoc Networks, pages 139–172. Addison-Wesley.

Levis, Philip, Lee, Nelson, Welsh, Matt, and Culler, David (2003). Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems*.

Park, Sung, Savvides, Andreas, and Srivastava, Mani (2001). Battery capacity measurement and analysis using lithium coin cell battery. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 382–387. ACM Press.

Perkins, C. (1997). Ad hoc on demand distance vector (AODV) routing.

Perkins, C., Belding-Royer, E., and Das, S. (2003). Rfc 3561 - ad hoc on-demand distance vector (AODV) routing.

S. Park, A. Savvides and Srivastava, M. B. (2000). Sensorsim : A simulation framework for sensor networks. In *the Proceedings of MSWiM 2000*.

Sundresh, Sameer, Kim, WooYoung, and Agha, Gul (2004). Sens: A sensor, environment and network simulator. In *The 37th Annual Simulation Symposium (ANSS37)*.

Szymanski, Boleslaw K. and Chen, Gilbert (2002). *Lecture Notes in Computer Science, Parallel Processing and Applied Mathematics: 4th International Conference*, chapter A Component Model for Discrete Event Simulation, pages 580–594. Springer-Verlag.

Ulmer, Craig. Wireless sensor probe networks - SensorSimII. http://www.craigulmer.com/research/sensorsimii/.

Xiang Zeng, Rajive Bagrodia, Mario Gerla (1998). Glomosim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulations*.