SENSE: A Sensor Network Simulator

Gilbert Chen, Joel Branch, Eugene Brevdo, Lijuan Zhu, and Boleslaw Szymanski
Department of Computer Science
Rensselaer Polytechnic Institute
110 8th Street,
Troy, NY 12180,
U.S.A.

February 3, 2004

Abstract

A new network simulator, named 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 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, fast inter-component communication, were also considered. More specifically, a memory-efficient packet management scheme was developed to minimize the amount of memory used for packet allocation, and an optimization technique was proposed to eliminate inter-component communication overhead.

1 Introduction

The emergence of wireless sensor networks created many open issues in network design [8]. The three main techniques for analyzing performance of wired and wireless networks traditionally 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 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 unsolved research, design and implementation problems, so measurements are virtually impossible. It appears that simulation is currently the only feasible approach to the quantitative analysis of sensor networks.

ns2 [5], perhaps the most widely used research network simulator, 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 interdependency between

modules. Such interdependency sometimes makes the addition of new protocol models extremely difficult and possible only by those who have intimate familiarity with the simulator. Being difficult to extend is 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 media, 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 under various circumstances.

Many other publicly available network simulators, such as J-Sim [4], SSFNet [7], Glomosim [2] and its descendant Qualnet [6], 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 terms of design and extensibility.

SENSE (SEnse Network Simulator and Emulator), described here, 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 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.

2 Design Philosophy

2.1 Extensibility, Reusability and Scalability

The enabling force behind the fully extensible network simulation architecture in SENSE is the recent progress in component-based simulation [14]. A *component-port model* frees simulation models from interdependency 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 needs.

The removal of interdependency 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 level of reusability can be made possible by the extensive use of C++ template: a component declared as a template class can handle different types of data.

Unlike many parallel network simulators, especially SSFNet [7] and Glomosim [2], parallelization is provided as an option to the users of SENSE. That reflects 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 have been doomed to be inefficient. Therefore, parallelizable models must require more effort than sequential models, while a good portion of users are not interested in parallel simulation at all. In SENSE, a parallel simulation engine can only execute 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.

2.2 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 have any programming skills. Extensibility and reusability are not their concerns. However, they require the simulation to be parallelizable.

The network builders are not satisfied with the available network templates, but they still depend upon 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 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 may need to modify available models or even build new ones from scratch. For example, they can develop a new MAC layer protocol and simply replace the original one with the new 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 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. There should be 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 follow in order for the models to be more accessible by others.

3 Component-Based Design

SENSE is built on top of COST [9], 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.

3.1 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 dependency 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 this is that the development of a component is now completely separated from the application context in which the component will be used, resulting in truly reusable components. Besides, components become more extensible, because there are fewer constraints on a component that provides certain 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.

3.1.1 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. We will present a technique in Section 6 to eliminate such communication overhead. Another problem 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.

3.1.2 Components for Sensor Network Simulation

The component-port model also gives the users a great deal of freedom in configuring sensor nodes. Figure 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. The inports and outports of the sensor are directly connected to the corresponding inports and outports of internal components. This structure, is changeable. The user can freely remove or add a component, as demanded by the particular goal of the simulation. An entirely new node structure can also be constructed on top of existing components or components built from scratch.

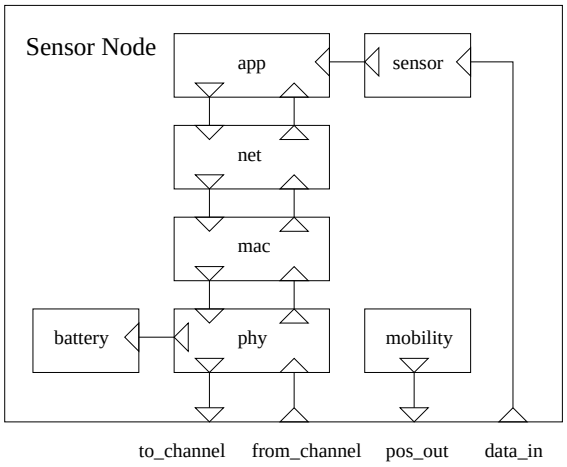

Figure 1: The internal structure of a typical sensor node

When configuring sensor nodes into a network, different configuration languages are available. Configuration involves setting the parameters of each component and then interconnecting their inports and outports. In this phase, components do not communicate with each other, so an 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 not preclude the possibility of using such languages as TCL or XML. In addition, it is quite possible to develop a simple scripting language specifically for the network configuration phase.

3.2 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 [14].

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 activated and then forced to process the future event.

Type III components are named autonomous components because they maintain their own simulation clock themselves. The *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 a 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. 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.

4 Packet Management

A network simulation is composed of two types of entities: one are the static components that simulate various network elements and the other are 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 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 programs that 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 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 that of a pointer, than a pointer can be kept, represented by dotted arrows; otherwise a direct copy, represented by solid arrows, will be more convenient.

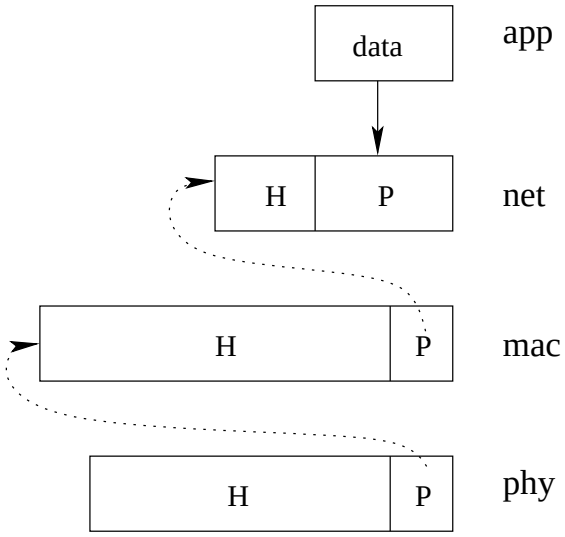

Figure 2: The Layered Packet Structure

Another decision we made regarding the packet management is that a packets sent by one node are 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 is considerable.

A standard programming technique, *reference counting*, is adopted to keep track of pack-

ets. 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 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 is 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" [1]. 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; however it does so 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;
  }
}
```

This 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();
  }
};
```

5 Component Repository

As the core design of SENSE is being finalized, we have 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. Three components, namely IEEE 802.11, AODV, and DSR, are the most complicated and consume a large portion of our development time, so each of them deserves a brief introduction here.

5.1 IEEE 802.11

The IEEE 802.11 component in SENSE implemented the distributed coordination function (DCF) described in the IEEE 802.11 standard [3]. When asked 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 SENSE implementation of IEEE 802.11 closely resembles that of ns2 [5]. 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, which greatly degrades the efficiency and readability.

5.2 AODV

Ad-hoc on demand distance vector routing (AODV) has been well-received as a routing protocol for MANETs. 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 [12] provides more details about AODV.

Our implementation in SENSE is based on the most current AODV internet draft [13]. 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 [13] for handling of unidirectional links have not been implemented. This is primarily because we only assume bi-directional links in our simulation. For similar reasons, 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.

5.3 DSR

The Dynamic Source Routing protocol for Mobile Ad Hoc Networks (DSR) [10] is an especially efficient implementation of on-demand routing for ad hoc networks. DSR provides the mechanisms of "Route Discovery" and "Route Maintenance"; these enable nodes to rapidly converge on recent changes in topology at need, without wasting energy and bandwidth at the physical layer on unneeded routing updates.

As of time of this writing, the DSR Routing Component for SENSE is under active development. Initial design is a complete rewrite of the simple base Flooding Routing component to satisfy DSR simulation parameters. The DSR component attempts to model the IETF MANET Internet Draft recommendations [11] for DSR as closely as possible, specifically by implementing the structure required to test any variation of the "Protocol Constants and Configuration Variables" as defined in section 9 of the draft [11].

Our initial implementation of DSR 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 homogeneous link layer environment. Moreover, we assume that the link layer provides acknowledgment for unicast packet transmission. 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. Immediate plans after finalizing the basic DSR component include support for promiscuous-mode operation, as the DSR definition takes advantage of this mode for passive data collection, and Route Maintenance tasks. Other plans include support for the optional DSR Flow State Extension, uni-directional links, and a data link layer which does not provide acknowledgment information for unicast packets.

6 Component Merging

As stated in Section 3.1.1, one problem associated with the component-oriented design is the inter-component communication overhead. For an event to pass between two components via a pair of an outport and an inport, several layers of function calls have to be involved. The C++ compiler cannot optimize away the overhead by treating these functions as 'inline' functions, because connection of ports is done during the runtime. To explain this point, let us consider a simple example illustrated in Figure 3.

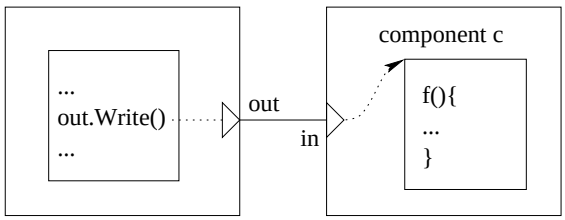

Figure 3: Communication between a Pair of Inport and Outport

Suppose there is a component c of class C that has an inport in. This inport is bound

to a member function f of c. An outport *out* of another component has been connected to the inport *in*. So when *out* is written by $out.Write()$, the $Write()$ function of the inport *in* will be called, which will in turn invoke the member function f of c.

The statement $out.Write()$ is equivalent to $c->f()$, so why not just replace the former by the latter? However, the C++ compiler cannot perform this optimization, since *out* is connected to *in* during the runtime by a $Connect()$ function that belongs to the COST API.

Yet we realize that it is possible to optimize a composite component, using a technique we refer to as *component merging*. A composite component is composed of several internal components (or subcomponents). The sensor node shown in Figure 1 is such a composite component. The connection between subcomponents within a composite component is usually static, making it possible to merge the subcomponents.

The transformation from a composite component into a primitive component must follow several rules:

- Each subcomponent must be converted to a nested class, with all inports and outports being removed.
- If an outport of a subcomponent is connected to an inport of the same or another subcomponent, then every write to this outport must be replaced by a call to the member function bound to the inport.
- If an outport of a subcomponent is connected to an outport of the composite component, then every write to this outport must be replaced by a write to the outport of the composite component.
- Every timer of any subcomponent must be moved to the composite component.
- Every inport of the composite component must be now bound to the member function that was bound to the inport of a subcomponent connected to the inport of the composite component.

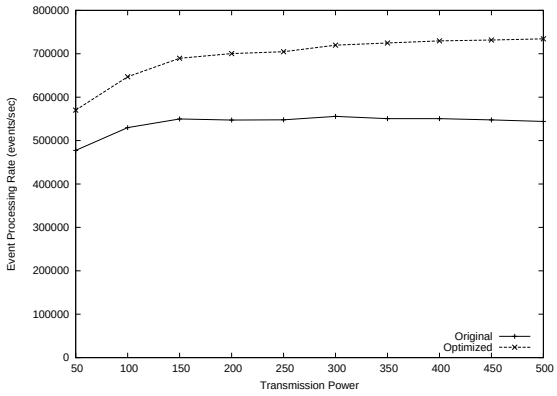

Figure 4: Performance Improvement with Component Merging

Figure 4 compares event processing rates before and after component merging is applied. The experiments were all conducted on a simulation of the flooding algorithm. The sensor

node structure is similar to that shown in Figure 1, except that there is no sensing data input. The application layer generates packets periodically, according to a constant bit rate model. The network layer is an implementation of the flooding algorithm, which re-broadcasts every received packet if the packet has reached its destination. For simplicity, a simple mac layer component is used, which is based on an unrealistic assumption that the bandwidth is infinite, so all transmissions are instantaneous and no collision would ever occur. The network consists of 100 nodes on a 5km by 5km territory, with the transmission power ranging from 50mW to 500mW. The conversion was done by hand. A performance improvement of roughly 30% can be clearly seen.

7 Conclusion and Future Work

The most significant feature of SENSE is its balanced consideration of modeling methodology and simulation efficiency. 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. We also proposed an optimization technique that can optimize away inter-component communication overhead. The simulator is therefore memory-efficient, fast, extensible, and reusable.

What is left to be done is to build 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. The visualization tool can also facilitate the configuration phase by allowing networks to be constructed graphically.

References

[1] Bjarne stroustrup's homepage. http://www.research.att.com/ bs/glossary.html.

[2] Global mobile system simulator. http://pcl.cs.ucla.edu/projects/glomosim/.

[3] Ieee 802.11, 1999 edition. http://standards.ieee.org/getieee802/802.11.html.

[4] J-sim. http://www.j-sim.org/.

[5] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[6] Qualnet. http://www.scalable-networks.com/.

[7] Scalable simulation framework. http://www.ssfnet.org/.

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

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

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

[11] David B. Johnson, David A. Maltz, and Yih-Chun Hu. The dynamic source routing protocol for mobile ad hoc networks (DSR), April 2003. Work in progress.

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

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

[14] Boleslaw K. Szymanski and Gilbert Chen. *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, 2002.