iDSRT: Integrated Dynamic Soft Real-time Architecture for Critical Infrastructure Data Delivery over WLAN*

Hoang Nguyen, Raoul Rivas and Klara Nahrstedt

University of Illinois at Urbana-Champaign
{hnguyen5, trivas, klara}@uiuc.edu

Abstract. Critical Infrastructures (CIs) such as the Power Grid play an important role in our lives. Of all important aspects of CIs, real-time data delivery is the most important one because appropriate decisions cannot be made without having data delivered in a timely manner. Also, the current trend for the real-time data delivery system, specifically SCADA (Supervisory Control and Data Acquisition) systems, is to move from a closed system toward an open system that has an open architecture and uses mature computer and communication technologies. Specifically, CIs tend to adopt micro-processor-based devices a.k.a. Intelligent Electronic Devices (IEDs) with commodity multi-threaded operating systems such as Linux-based OSs and COTS wireless LAN technology such as 802.11 WLAN. However, these trends pose a question of whether real-time guarantees, using these technologies, are feasible and can be practically implemented without many modifications of the commodity platform.

In this paper, we present a design and implementation of iDSRT, a system that provides end-to-end soft real-time data delivery guarantees over 802.11 WLAN with minimal changes of the operating system and no changes in the MAC layer. iDSRT consists of three important and integrated schedulers: task scheduler, packet scheduler and node scheduler to achieve the goal. The integration requires a coordination mechanism among these components. We formulate this coordination problem as a convex optimization problem that can be solved using standard convex optimization techniques. We implement iDSRT in Linux and evaluate it in an experimental testbed. The results are promising and show that iDSRT can successfully achieve soft real-time guarantees with very low packet loss rate compared to 802.11 best-effort systems.

1 Introduction

Critical Infrastructures (CIs) such as the Power Grid have gained significant attention recently due to their crucial roles in our lives. Among all of the important aspects of CIs, the real-time monitoring & control system, specifically SCADA (Supervisory Control and Data Acquisition) systems, is the most important one because appropriate decisions cannot be made without having data delivered in a timely manner. Therefore, previous SCADA systems have been built using proprietary and closed system components specifically to achieve the real-time requirement. However, due to the cost effective reasons, the current trend of SCADA is to move from a closed system toward an open system that has open architecture using wide-range of services provided by mature computer and communication technologies such as TCP/IP, wireless LAN, Linux operating systems and middleware.

Specifically, there are two interesting trends of SCADA. The first trend is to move from micro-controller-based devices toward micro-processor-based devices a.k.a. Intelligent Electronic devices (IEDs). Furthermore, as these devices get more intelligent and responsible for monitoring, control and management functions, their operating systems will become multi-threaded. The obvious benefits of this trend are to be able to use available services in modern operating systems such as the TCP/IP networking stack and standard security features. The second trend is to deploy Wireless

* This material is based upon work supported by the National Science Foundation under Grant CNS-0524695 and Vietnam Education Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of those agencies.

LAN (WLAN) in SCADA due to the success of recent wireless technologies such as 802.11 or 802.15.4 [1][3][6]. Furthermore, the wireless solution also has certain advantages over its wire-line counter-part such as the easy and extended device deployment, low deployment cost, and ease of reconfiguration. Most importantly, in some environments that are physically dangerous and have geographically difficult locations (e.g. power substations), those benefits are significantly amplified.

The general use of WLAN is shown in Figure 1. The scenario includes both real-time monitoring/control and non real-time management applications. Intelligent Electronic Devices (IEDs) periodically send sampling measurements (such as voltage, current, temperature) to a *gateway*. The gateway collects and processes the measurements, and issues necessary control actions to IEDs. The time requirement in this scenario may be as fine-grained as in the order of milliseconds [7]. In addition to the *real-time monitor and control* functionality, both the gateway and IEDs need to handle other *management* tasks. For example, the gateway may upload a configuration file to IEDs via a secure protocol (e.g. SSL).

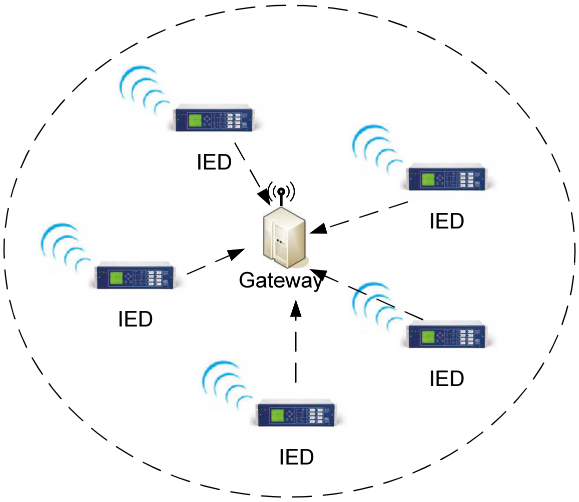

Fig. 1. Wireless LAN deployment in a Power substation

Despite of many benefits coming from the current trends, it is unclear whether we can build a soft real-time system [1] on top of such commodity operating systems and wireless MAC. Specifically, the question is on the feasibility of building *a system that can provide control of end-to-end delay (i.e. OS delay and wireless network delay) guarantees over 802.11 wireless MAC with minimal or no changes of the operating systems and wireless MAC*. Interestingly, this problem, in fact, turns out to be very challenging because the end-to-end delay guarantee requires the delay control at both the operating system and the wireless network system. Even if we have the real-time operating system such as [2][9][24][26][30][22][34] and the real-time network such as real-time MAC [28][16][17], the end-to-end delay cannot be guaranteed if the OS and the network do not coordinate with each other. For example, if the operating system keeps finishing the task near the end of task's period and leaves no time for the network to transmit then the task will keep missing deadlines.

In this work, we show a design and implementation of iDSRT which goal is to provide fine-grained end-to-end delay guarantees over single-hop wireless networks. The principle design of iDSRT is to require minimal modification or support from hardware such as network interface card (NIC) and from the operating system. It is designed to work on top of 802.11-compatible NIC and commodity Linux operating system. iDSRT integrates three schedulers: task scheduler, packet scheduler and node scheduler to achieve the fine-grainted end-to-end delay guarantees. Specifically, it employs EDF

[1] We are interested in soft real-time systems since within the considered CIs the time guarantees are in millisecond, seconds and minutes with few delivery violations.

(Earliest Deadline First) scheduling algorithm for both the *task scheduler*, called DSRT (Dynamic Soft real-time CPU scheduler), and the *network scheduler*, called iEDF (Implicit EDF). The coordination between these two schedulers is executed by a Coordinator entity, called iCoord, sitting at the middleware layer. The Coordinator coordinates the access to the shared wireless medium, hence plays the role of the *node scheduler*. iDSRT decouples the dependency between DSRT and iEDF by decomposing the real-time task into the CPU task and the network task with appropriate deadline assignment. The deadline assignment is done in the way that the total stress factor on the system is minimized.

In summary, our contributions in this paper are:

1. formulating the deadline assignment problem as an convex optimization problem which objective function is to minimize the total stress factor on the system.
2. a design and implementation of an integrated system providing soft real-time end-to-end delay guarantees with minimal changes in the commodity Linux operating system and no changes in MAC layer.
3. a performance study of iDSRT in an experimental test-bed of wireless nodes.

The rest of the paper is organized as follows. Section 2 presents our system model, notations and assumptions. In Section 3, we show the architecture of iDSRT and an overview of its components. Section 4 shows our approach to this problem. Section 5, Section 6 and Section 7 give the details of iCoord, DSRT and iEDF. Section 8 presents necessary details of iDSRT implementation. In Section 9, we show our evaluation of iDSRT. Section 10 gives the related work and finally, Section 11 concludes the paper.

2 Models and Definitions

In this section, we show the models, definitions, notations and assumptions used in this work. Table 1 summarizes notations used in this paper.

2.1 Network Model

We consider a single-hop wireless network model where each node can hear all other nodes as shown in Figure 1. There are n clients (i.e. IEDs) $N_1, N_2, .., N_n$ in the network and a node S acting as a server (i.e. gateway). Formally, the network is modeled as an undirected graph $G = (V, E)$ where $V = \{N_1, .., N_n, S\}$ and $E = \{(N_1, S), ..., (N_n, S)\}$.

Each client N_i has m_i ($m_i \geq 0$) real-time (RT) applications/streams and may have best-effort (BE) applications/streams running simultaneously. RT applications stream the data from the client to the server [2] (see subsection 2.2). A typical example of a RT application/stream is the SCADA monitoring application, where sampled power-related data (e.g. frequency, amplitude, angle, voltage) is processed and sent to the server over the wireless network in real-time. Each RT application/stream has to conform to its QoS specification in terms of end-to-end delay (EED) requirement.

EED is the sum of the delay at the sending side (i.e. at the client side), the propagation delay and the delay at the receiving side (i.e at the server side). Controlling any of these components will affect EED. Our system, however, *only controls the delay at the sending side.* We assume the propagation delay is negligible compared to other two delay components. Even if the propagation delay is non-negligible, we cannot control this delay component and thus is not central to our study. Furthermore, the receiving delay incurred at the gateway, including computation delay and MAC transmission delay, is small too. The reason is that we assume the gateway is a device with powerful computation and communication capabilities compared to the clients. Hence, controlling of this small delay component does not have much effect on the EED and it is also not the focus of our study.

The sending delay consists of the computation delay incurred by the the OS scheduling and the communication delay incurred by the network scheduling. This delay component can be controlled

[2] Even though not specified explicitly in the model, the extension to support communication between two clients can be done in a similar manner.

by assigning deadlines to the computation and communication sub-tasks at each client(see Section 2.2). As long as these sub-tasks are finished on time by the OS scheduler and network scheduler, the EED requirements can be met.

In our model, the BE applications may or may not stream the data to the server. These BE applications, if not monitored and enforced properly, can affect the QoS performance of other RT tasks since they are not aware of real-time aspects of the system. Some typical BE applications in CIs are FTP application for downloading/uploading devices' configuration or data encryption/decryption for secure communication. These network-intensive and computation-intensive applications may consume exhaustively network and CPU resources in the system if not constrained.

2.2 Task Model

We model the RT streaming applications (e.g. monitoring applications) as RT networked tasks, at the client side, composed of the computation and communication sub-tasks. The end-to-end delay requirement of streaming applications is now transformed into the *end-to-end deadlines* of the RT networked tasks used for scheduling.

Formally, we denote A_{ij} for the jth RT networked task/application on the client N_i[3] where $i = 1..n$, $j = 1..m_i$. We also denote A_S as the networked task running on the server S. Each task A_{ij} has a period P_{ij}. It has two sub-tasks A_{ij}^{CPU} and A_{ij}^{Net} that needs to be processed in order (see Figure 2). That means, within period P_{ij}, the sub-task A_{ij}^{CPU} needs C_{ij} time unit for sampling and processing data. After the data gets processed, the sub-task A_{ij}^{Net} needs R_{ij} time unit to send it to the server task A_S on server S over the wireless network G. The deadline D_{ij} of task A_{ij} is equal to the period P_{ij}.

Both C_{ij} and R_{ij} are CPU and network resources consumed in time. C_{ij} is calculated by the number of consumed cycles over the CPU frequency. We assume the frequency of the CPU is fixed. Similarly, R_{ij} is the time of task A_{ij} and its underlying OS/network protocol stack to transmit a packet of size PS_{ij} bytes over the wireless MAC with measured bandwidth B_{ij} at node N_i, i.e. $R_{ij} = PS_{ij}/B_{ij}$ to the server S.

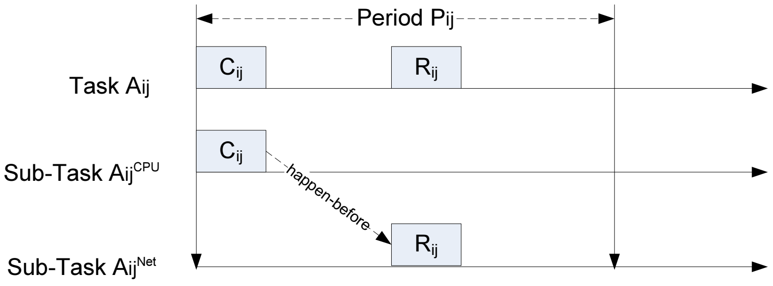

Fig. 2. Task Model

3 iDSRT Framework

Our first goal is to design/establish a scheduling and coordination framework of three important schedulers (i.e. the task scheduler, the packet scheduler and the node scheduler) that deliver end-to-end soft real-time guarantees in system G. The next goal is to increase the compatibility of the system. Specifically, the system should be able to run on a commodity platform such as commodity Linux-based operating system and 802.11 MAC layer. Each node N_i will consider time-sensitive

[3] The terms "RT application A_{ij}" and "RT task A_{ij}" are used exchangeably.

Notation	Description
n	number of clients
N_i	client i
m_i	number of real-time tasks/applications on client N_i
S	the server
$A_{ij}(C_{ij}, R_{ij}, P_{ij})$	the jth real-time task/application A_{ij} running on client N_i
P_{ij}	period of real-time task/application A_{ij}
D_{ij}	deadline of real-time application A_{ij} and $D_{ij} \leq P_{ij}$.
C_{ij}	number of time unit A_{ij} consumes CPU resource to process its data.
PS_{ij}	packet size of application A_{ij}
B_{ij}	bandwidth of the wireless MAC of client N_i
R_{ij}	number of time unit A_{ij} sends one packet of size PS_{ij}. $R_{ij} = PS_{ij}/B_{ij}$
A_S	server application running on gateway S
$A_{ij}^{CPU}(C_{ij}, D_{ij}^{CPU}, P_{ij})$	CPU sub-task of A_{ij} with computation time C_{ij}, deadline D_{ij}^{CPU} and the period P_{ij}.
$A_{ij}^{Net}(R_{ij}, D_{ij}^{Net}, P_{ij})$	network sub-task of A_{ij} with transmission time R_{ij}, deadline D_{ij}^{Net} and the period P_{ij}.
T_{ij}	a deadline assignment to the partitioned tasks of A_{ij}, $T_{ij} > 0$ & $T_{ij} < D_{ij}$
T	a deadline assignment for the whole task set A_{ij}, i.e. $T = \{T_{ij}\}$

Table 1. Table of notations

scheduling of a) RT tasks $A_{ij}, i = 1..n, j = 1..m_i$ under competition of best-effort tasks, b) network packets of connections belonging to the RT networked application A_{ij} and BE tasks at the node N_i and c) node N_i with respect to other nodes $N_k, k = 1..n, k \neq i$ due to the shared access to wireless medium.

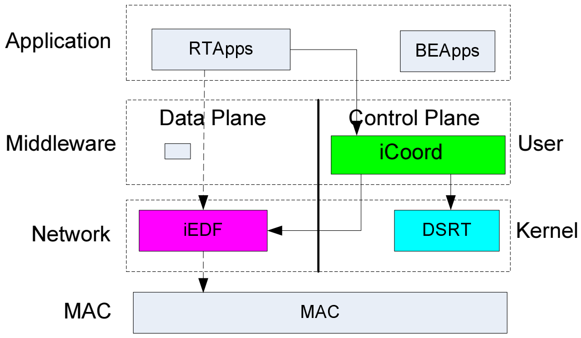

Fig. 3. End-to-End Integrated Dynamic Soft Real-time Framework (iDSRT)

The scheduling and coordination framework resides in the middleware, network and OS layers as shown in Figure 3 and it is called *iDSRT*. It allows RT applications and BE applications to run together and share resources in controlled manner. RT applications rely on iCoord (Integrated Coordination) - a distributed middleware component residing in the control plane of the protocol stack. It receives QoS specification from RT applications, performs RT application profiling, and does the QoS negotiation on behalf of the RT applications A_{ij}. Its central role is coordinating resource management components within each node N_i and among nodes $N_i, i = 1..n$ and S in G to ensure end-to-end delay guarantees. Section 5 describes the details of iCoord.

Any potential conflicts among RT tasks $A_{ij}, j = 1..m_i$ and BE tasks on node N_i are resolved by the Dynamic Soft-Real-time CPU Scheduler, called *DSRT* [22]. DSRT guarantees CPU resources for RT applications by using an adaptive EDF scheduling algorithm. It is “soft” because it does not manage other resources of the hardware and thus does not prevent the preemptions due to non-CPU hardware interrupts. However, the soft guarantees are within the timing bounds of SCADA tasks. Section 6 will give more details.

The last component in the iDSRT framework is the iEDF (Implicit Earlier Deadline First) packet scheduler. Essentially, iEDF is a network packet scheduler residing on-top of the MAC layer. It takes the implicit contention approach to schedule transmission slots according to the EDF policy. It manages the packet queue of each node and makes sure all nodes agree on the same packet to transmit over the shared medium within a specific time slot. Section 7 will discuss more details.

4 Deadline Assignment Problem and Solution

4.1 Deadline Assignment Problem

In this section, we formulate the “deadline assignment” problem. As described in the previous section, we employ the EDF algorithm for CPU scheduler (DSRT), taking care of the task scheduling. Similarly, the EDF algorithm is also used in the network scheduler (iEDF), taking care of the packet scheduling (i.e., intra-node scheduling), and the node scheduling (i.e., inter-node scheduling). These two schedulers (DSRT and iEDF) must coordinate with each other so that the end-to-end deadline of RT applications A_{ij} can be met. The approach we take is partitioning the end-to-end deadline into sub-deadlines for the CPU scheduler and network scheduler. Thus, as long as the CPU scheduler and the network scheduler can schedule the sub-tasks correctly, the end-to-end deadlines will be guaranteed.

To illustrate the deadline assignment problem and the basic idea of how it is used to solve the end-to-end delay guarantee, let us consider scenarios in Figure 4. The original RT task $A_{ij}(C_{ij}, R_{ij}, P_{ij})$ on node N_i can be thought as being split into two sub-tasks: the CPU task $A_{ij}^{CPU}(C_{ij}, D_{ij}^{CPU}, P_{ij})$ and the network task $A_{ij}^{Net}(R_{ij}, D_{ij}^{Net}, P_{ij})$, where D_{ij}^{CPU} and D_{ij}^{Net} are the relative deadlines for CPU task and network task and $D_{ij}^{CPU} + D_{ij}^{Net} = D_{ij}(\leq P_{ij})$. Furthermore, if the phase of the network task A_{ij}^{Net} is equal to D_{ij}^{CPU} (i.e. the relative deadline for the CPU task), then the end-to-end delay of A_{ij} can be met as long as the CPU task and the network task meet their deadlines within the node N_i.

A deadline assignment $T = \{T_{ij} | T_{ij} > 0 \ \& \ T_{ij} < D_{ij}, i = 1..n, j = 1..m_i\}$ is a set of deadlines T_{ij} assigned to each corresponding RT task A_{ij} i.e. $D_{ij}^{CPU} = T_{ij}$ and $D_{ij}^{Net} = D_{ij} - T_{ij}$. T is *valid* if it yields a feasible scheduling for the CPU task set at each node and the network task set in the system. Furthermore, in addition to the validity constraint, T can also be optimized according to an objective function. Different objective functions may lead to different ways to assign deadlines and thus, different solutions [18][19][31][29]. In other words, a solution of the deadline assignment problem T specifies a set of $\{T_{ij}\}, u = 1..n, j = 1..m_i$ where 1) assigning $D_{ij}^{CPU} = T_{ij}$ and $D_{ij}^{Net} = D_{ij} - T_{ij}$ will yield a feasible scheduling for all CPU tasks and network tasks and 2) T is optimized according to an objection function. In the next section, we describe our approach to this problem.

4.2 Solution to Deadline Assignment Problem

Intuitively, for any deadline assignment $T = \{T_{ij} | T_{ij} > 0 \ \& \ T_{ij} < D_{ij}, i = 1..n, j = 1..m_i\}$, decreasing T_{ij} will put more stress on the CPU of node N_i and less stress on the network and vice versa. Thus, it is desired to put the stress fairly on both resources. Formally, let us define the stress factor on CPU resource of a deadline assignment T_{ij} as

$$F_{ij}^{CPU}(T_{ij}) = \frac{C_{ij}}{T_{ij}}$$

and similarly for the network resource as [4]

[4] Note that $D_{ij}^{Net} = D_{ij} - T_{ij}$

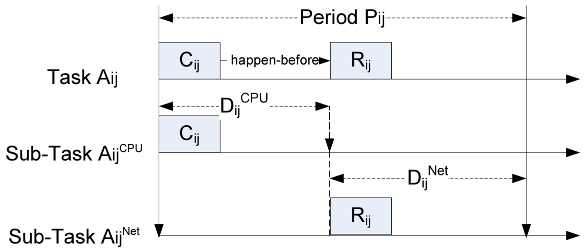

Fig. 4. Illustration of deadline assignment problem for the task A_{ij}

$$F_{ij}^{Net}(T_{ij}) = \frac{R_{ij}}{D_{ij} - T_{ij}}$$

and the total stress of a deadline assignment T_{ij}

$$F_{ij}(T_{ij}) = F_{ij}^{CPU}(T_{ij}) + F_{ij}^{Net}(T_{ij}).$$

To give an intuition of why this stress factor is important, let us compare the assignment T with the the optimal case where CPU sub-task and network sub-task have no dependencies and thus can be executed in any order. Without any dependencies between CPU sub-task and network sub-task, the stress factor of a task A_{ij} is

$$F_{ij}^{*} = \frac{C_{ij}}{D_{ij}} + \frac{R_{ij}}{D_{ij}}.$$

It is easy to see that

$$F_{ij}(T_{ij}) = \frac{C_{ij}}{T_{ij}} + \frac{R_{ij}}{D_{ij} - T_{ij}} > \frac{C_{ij}}{D_{ij}} + \frac{R_{ij}}{D_{ij}} = F_{ij}^{*}. \tag{1}$$

.

The above equation means that due to the dependency among sub-tasks, the resource partitioning always puts more stress on both resources. Thus, it is preferable to minimize the total stress of all tasks on the system over all possible deadline assignment T

$$\text{minimize } F(T) = \textstyle\sum_{i=1}^{n} \sum_{j=1}^{m_i} F_{ij}(T_{ij}), \forall T$$

.

$F(T)$ is used as the objective function to the deadline assignment problem. We now formulate the constraints from the CPU scheduling and network scheduling.

The CPU scheduler and the network scheduler need an EDF admission control, where tasks have deadlines less than periods. This can be done by using the processor demand criteria [14][10] (cf. Sections 6 and 7). Essentially, the basic idea is to test whether within every possible time interval L, the processor demand of all tasks having deadline less than L is less than the time length L.

Therefore, the problem can be formulated as an optimization problem as follows.

$$\begin{gathered}\text{minimize } F(T) \\ \text{variables } T = \{T_{ij}, i = 1..n, j = 1..m_i\} \\ \text{constraints } C_{ij} < T_{ij} < D_{ij},\ i = 1..n, j = 1..m_i \\ \text{schedulability tests for CPU and network tasks}\end{gathered}$$

The above optimization problem is a non-linear optimization problem because the objective function is non-linear. However, it is easy to see that the objective function is a convex function because it is the sum of convex functions $F_{ij}(T_{ij})$ [11]. Furthermore, except the schedulability tests for CPU and network tasks, all other constraints are indeed linear (and hence, convex). We now are interested to see whether the schedulability tests are actually convex.

In [12], the authors analyze the processor demand criterion and come up with the concept of "space of EDF feasible deadlines". Furthermore, they give a method to compute the exact EDF feasible region. Even though this exact region is nice and interesting, it is still complicated and thus not quite useful for applications. Therefore, they provide a simpler method to find an approximation of this space and interestingly, it turns out this approximation is convex! Thus, the schedulability tests can be approximated and become convex.

Now, because the objective function and all constraints are convex, the optimization problem becomes a convex optimization problem. Therefore, by using standard convex optimization solving techniques such as Lagrange multipliers method [11], the problem can be solved theoretically and numerically. It is important to note that solving convex optimization is *tractable*, i.e. it has polynomial complexity. We omit the details because they can be found in convex optimization [13] or non-linear programming literature[11]. Once solved, the result of this optimization problem is the deadline assignment $T = \{T_{ij}, i = 1..n, j = 1..m_i\}$ assigned to each corresponding task A_{ij} (i.e. $D_{ij}^{CPU} = T_{ij}$ and $D_{ij}^{net} = D_{ij} - T_{ij}$).

5 Integrated Middleware Coordination (iCoord)

iCoord is a distributed middleware component which coordinates all system scheduling components to ensure RT applications meet their deadlines. It operates in the control plane of the node's protocol stack to provide the node registration service, task profiling and coordination services. Its services are a set of middleware libraries whose computation overhead is charged to the calling tasks's computation.

Figure 5 shows the middleware control architecture of iCoord. iCoord consists of two modules: *Local iCoord* residing on each client N_i and *Global iCoord* residing on the gateway S. Local iCoord is in charge of coordinating system components at each node N_i and communicates with Global iCoord to assist in inter-node scheduling with other nodes' Local iCoord(s). Global iCoord executes global services on server S, where Local iCoord executes local services on each client N_i. Together, they ensure distributed utility services, such as the coordination service, registration service and the profiling service.

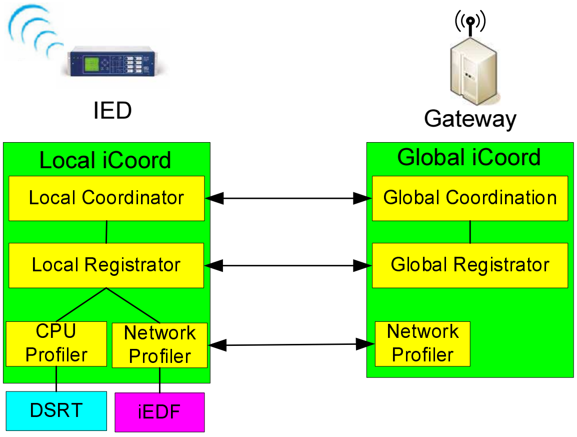

Fig. 5. Middleware control plane architecture iCoord

Figure 6 summarizes the protocol within iCoord.

Registration Service is a service that takes care of the registration of real-time applications. Essentially, every RT application has to register with iDSRT because un-registered applications are

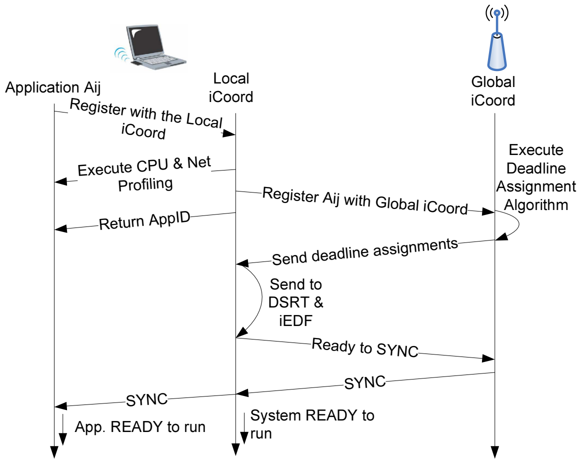
Application Aij
Local iCoord
Global iCoord
Register with the Local iCoord
Execute CPU & Net Profiling
Register Aij with Global iCoord
Execute Deadline Assignment Algorithm
Return AppID
Send deadline assignments
Send to DSRT & iEDF
Ready to SYNC
SYNC
SYNC
App. READY to run
System READY to run

Fig. 6. iCoord protocol

treated as BE applications. First, the registration is done via the *Local iCoord Registrator*. The registration request from an RT application A_{ij} includes

- Tuple of $(pid, saddr, sport, daddr, dport)$
  where parameter pid, $saddr$, $sport$, $daddr$, $dport$ are the process identifier, the source address, the source port, the destination address, the destination port respectively. These parameters are used to uniquely identify each real-time communication application A_{ij}.
- Period $P_{ij}(\mu s)$
- Requirements $C_{ij}(\mu s)$ on CPU resource and network resource $R_{ij}(\mu s)$, which are measured by the profiling services.

The Local iCoord Registrator sends the registration information of this application to the *Global iCoord Registrator*. After the Global iCoord Registrator acknowledges the successful registration of the application A_{ij}, the Local iCoord Registrator returns a unique ID calculated from the tuple of registration information to the application. Finally, the Local iCoord Registrator invokes the CPU and network profiling services to approximate the CPU and network usage of the application (i.e. C_{ij} and R_{ij}). Finally, it sends the profiles of this task to the Global iCoord Registrator so that the node admission control, inter-node scheduling and coordination can be performed.

Profiling Service consists of the CPU profiler and network profiler on each client N_i. These two profilers are invoked after the registration phase. The CPU usage is measured by having DSRT run several instances of the RT task A_{ij}. Similarly, the network profiling is done by measuring the packet round-trip-time between the networked application at the client N_i and the server S.

Coordination service is a distributed middleware component. Similar to the Registration service, Coordination service has a Global Coordinator at the gateway S and a Local Coordinator at each node N_i. The Global Coordinator at the gateway gathers profiles of all RT applications from the Global Registrator and performs the deadline assignment algorithm discussed in Section 4. Then, it sends this information to all *Local iCoord Coordinators*. The information includes deadline assignments for the inter-node (i.e. D_{ij}^{Net}) and intra-node (i.e. D_{ij}^{CPU}) scheduling of all the tasks in the system G.

Upon receiving the deadline assignment of all tasks, the Local Coordinator confirms with DSRT and iEDF about the acceptance of these local tasks. At the end of this phase, each Local iCoord Coordinator notifies the Global iCoord Coordinator that node N_i is ready, and all local components DSRT, iEDF and Local iCoord wait for the SYNC message from the Global iCoord Coordinator.

In the last phase, the Global Coordinator waits for all acknowledgments from Local Coordinators and broadcasts the SYNC message. The SYNC message start the run-time of the whole system G.

6 DSRT (Dynamic Soft Real-time Scheduler)

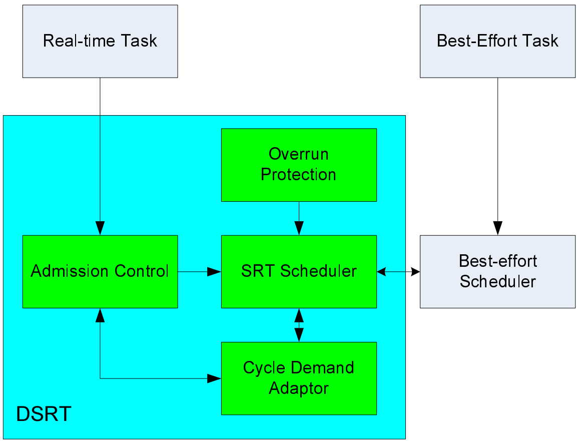

Fig. 7. DSRT Architecture

DSRT is responsible for CPU task scheduling according to their deadlines. Specifically, on client N_i, it manages real-time CPU tasks $A_{ij}^{CPU}, j = 1..m_i$ as modeled in section 2.2. To achieve this objective, DSRT is composed of three basic components, the Admission Control, the Earliest-Deadline-First (EDF) Scheduler and the Cycle Demand Adaptor.

On a node N_i, before using the realtime capabilities of the system, a new RT task A_{ij}^{CPU} must register itself with iCoord as a RT task in the DSRT. Specifically, it must specify its period, its worst case execution time and its relative deadline [5]. The admission control for DSRT on a node N_i is the EDF schedulability test. It means,

$$\forall L \in DLset, L \geq \sum_{j=1}^{m_i} (\lfloor \frac{L - D_{ij}^{CPU}}{P_{ij}} \rfloor + 1) C_{ij} \tag{2}$$

where $DLset = \{d_{kl} | d_{kl} = lP_{ik} + D_{ik}^{CPU}, 1 \leq k \leq m_i, l \geq 0\}$ is the set including all tasks' deadlines less then the hyper-period of all periods (i.e. least common multiplier of $P_{i1}, .., P_{im_i}$).

If the condition is met, the task A_{ij}^{CPU} is added to the running queue of the EDF Scheduler and is scheduled to run in the next period. If the task cannot complete its job in the allotted time, due to demand cycle variations, the *Overrun Timer* will preempt the task to best-effort mode. In this case, the task A_{ij}^{CPU} will only be allowed to run after all other real-time tasks have used their allotted CPU time. The Overrun Timer removes the task from the running queue and adds it to the overrun queue. Tasks in best-effort mode compete against each other and use the standard OS non-realtime scheduler (Linux in the case of our implementation). Therefore, they cannot get a guaranteed CPU allocation.

If the deadline D_{ij}^{CPU} is not met, the Cycle Demand Adaptor will keep track of this event. If it detects that the change in the cycle demand is persistent and that assigned deadlines are not met, it will try to increase the allotted cycle demand for this particular task A_{ij}^{CPU}. In that case the Cycle Demand Adaptor will query the DSRT admission control to verify whether there are enough CPU resource to increase the allotted resource for the task A_{ij}^{CPU}.

[5] The information C_{ij} and D_{ij}^{CPU} is provided by iCoord as explained in Section 5.

7 iEDF (Implicit Earliest Deadline First packet scheduler)

iEDF is a distributed network scheduler that takes "implicit contention" approach to perform the EDF packet scheduling algorithm [15][16]. Each client uses iEDF as its network scheduler. Conceptually, this network scheduler is actually an *outgoing-packet scheduler* working on top of the MAC layer. It manages how packets are prioritized to ensure they will meet the deadlines. Technically, it is a kernel-loadable queue-management module hooking into basic network operations of a particular NIC device. More technical information will be given in Section 8.2. Also, it is important to emphasize that we are focusing on the network sub-task A_{ij}^{Net} processing network packet of the RT application A_{ij} and thus in this section "real-time task" refers to the "network sub-task".

The basic idea of implicit contention scheduling is to have each client run the same task scheduling algorithm on the same set of RT tasks. At any time slot where all clients need to agree on who can access the shared wireless resource according to the Deadline Assignment solution (Section 4.2), each client executes the same scheduling algorithm on the current set of RT tasks [6]. Because all nodes run the same scheduling algorithm on the same input, the selected RT task is identical. Once a RT task is selected, only the client having the selected RT tasks can access the shared wireless resource. Once the RT task finishes its access to the shared resource, all clients update the current set of real-time tasks. In this manner, clients "*implicitly*" agree on the one to exclusively access the shared resource without exchanging any information during their network operation. All the information about RT tasks are exchanged during the bootstrapping.

iEDF is an implicit contention scheduling which uses EDF as the packet scheduling algorithm. At any time slot, all clients agree on a RT task A_{ij}^{Net} to access the shared wireless medium according to the EDF policy. Specifically, for a client N_i, RT tasks $A_{ij}^{Net}, j = 1..m_i$ running on N_i are called *local RT network applications* and other RT applications running on other clients are called *remote RT network applications.* iEDF at each client N_i maintains the deadline assignment and task information of remote RT network tasks in addition to its local RT network tasks disseminated via iCoord (see Section 5).

Once iEDF has all network task deadline information, it creates a "shadow network task" for each remote network task. The shadow network task has the same period, deadline and transmission time as the network task being shadowed. When the shadow network task $A_{kj}^{Net}, j = 1..m_k$ "executes" on $N_i, i \neq k$, it does nothing but sets up a timer to wake up after the transmission of A_{kj}^{Net}. On waking up, the shadow network task again notifies iEDF that the remote network task is supposed to finish now. On this event, iEDF schedules another RT network task, either local or remote (shadow) for the next transmission. In this way, iEDF is doing the EDF scheduling algorithm in a distributed manner. Furthermore, packet collisions will rarely happen because iEDF at each client aims to ensure and comply to the global deadline assignment. Figure 8 shows an illustration of this implicit contention.

Even though the principle of iEDF is very simple, there are couple of issues that we need to address. The first issue is the correct estimation of the transmission time of the shadow network task. For any particular transmission, the remote network task A_{kj}^{Net} may finish earlier than expected due to worst case profiling and estimation of R_{kj}. It may also finish later than expected due to the noisy and unreliable channel. In the former case, iEDF ignores the early transmission and accepts the waste of idle network resource. In the latter case, iEDF actually has to avoid starting another transmission to minimize the packet collisions. To resolve this issue, iEDF only needs to over-hear the wireless network to know when the remote network task finishes. This is a very simple solution yet good enough to resolve the scheduling issues.

The second issue is that even though iEDF is a network scheduler, it still needs non-negligible CPU resource for network tasks (local and remote). To resolve this issue, we let the network task's CPU consumption to be charged to the computation time of the corresponding RT applications.

Last but not least, a very important issue is the admission control. It is obvious that iEDF is a non-preemptive EDF scheduling because a network transmission cannot be preempted. Unfortunately, it has been known that non-preemptive EDF is not optimal and in fact, no known optimal

[6] Note that this is perfectly reasonable assumption for the application domain that we are considering. The SCADA IED sensing devices are running the same software and the same set of RT tasks

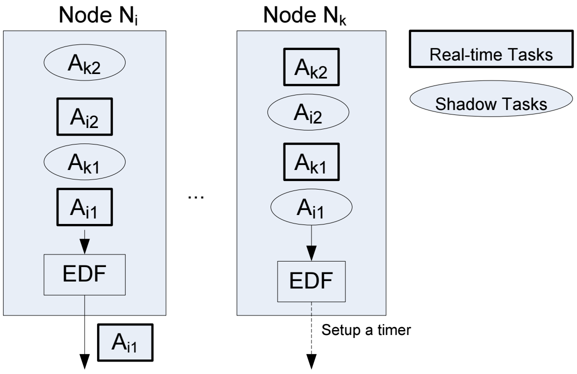

Fig. 8. Illustration of implicit contention scheduling

solution to the non-preemptive scheduling problem is computationally tractable anyway [14]. However, non-preemptive EDF still remains universal for periodic tasks (and sporadic tasks) [14][23]. In [23], Jeffay et al. showed a necessary condition of non-preemptive EDF scheduling done in pseudo-polynomial time and can be used for the admission control purpose. In iDSRT architecture, the network admission control is performed at $iCoord$.

8 Implementation

We have implemented iDSRT in nodes running Linux with kernel version 2.6.16 and it is compatible for later kernel versions. The overall objective of our implementation is to minimize the changes in the kernel. In this section, we describe details of the implementation of each component in iDSRT.

8.1 DSRT Implementation

DSRT was originally implemented by Chu et al. [22] in Linux Kernel 2.4. However, due to incompatibilities to Linux Kernel 2.6, DSRT is implemented from scratch in Linux Kernel 2.6. Our implementation has a series of kernel modules and patches taking advantage of the new features provided by the Linux Kernel 2.6.

DSRT implements nine new system calls allowing RT tasks to communicate with DSRT. These system calls provide DSRT with the information required to reserve CPU resource and prioritize a task according to its QoS requirements. In these system calls the task A_{ij}^{CPU} specifies the average cycle demand, which is used to calculate C_{ij}, (dividing by the CPU frequency), the deadline D_{ij}^{CPU} and the period P_{ij}. DSRT provides information about the performance and the status of the RT task, including the number of times a task tried to overrun and the statistical CPU utilization. The list of system calls is shown in the Table 2.

Our DSRT implementation needs only one kernel patch. That is the patch on the file *sched.c* in order to provide CPU accounting for each task. Linux currently provides such mechanism in the kernel but only with maximum resolution of 1 *jiffy* (number of iterations of the kernel per second)[7] while we need high precision task accounting to the microsecond resolution. Simply increasing the *jiffy* resolution will cause enormous kernel overhead. In our implementation, we measure CPU usage of real-time tasks in cycles instead of jiffy. This is achieved by adding a hook in $schedule()$ function. This hook is called every time that a context switch is about to occur. It allows us to measure the elapsed cycles between the current and the previous context switch and therefore precisely account for the CPU time of each task. Once done, the number of cycles is converted to time unit by dividing the number of cycles by CPU frequency.

[7] Within Linux 2.6.10, a jiffy is by default 4ms.

System Call	Description
cpu_request_profile(period)	Requests online cpu profiling to the DSRT.
cpu_begin_profile()	Signals the DSRT that the task is ready to begin the profiling of a new job.
cpu_finish_profile()	Signals the DSRT that the task completed the job for that period.
exit_srt(usage_statistics)	Unregister the task as realtime. The DSRT provides task accounting statistics.
enter_srt(period, deadline, avgcycles)	Requests soft realtime guarantees for the task.
begin_job()	Signals the DSRT that the task is ready to process a new job.
finish_job()	Signals the DSRT that the task completed the job for that period and yields the CPU.
exit_srt(usage_statistics)	Unregister the task as realtime. The DSRT provides task accounting statistics.
synchronize_period()	This function sets the beginning of the period. It sets the next deadline of the job to the current time plus the length of the period.

Table 2. List of system calls implemented in DSRT

We implement the rest of the DSRT as a kernel module. We use high-resolution timers provided in the kernel to ensure that tasks can wake-up at the precise time and to prevent overruns from greedy BE and RT tasks.

DSRT has a new data structure to store the QoS parameters $C_{ij}, D_{ij}^{CPU}, P_{ij}$ of the task containing information about the state of the RT task used by both the EDF scheduler and the Cycle Demand Adaptor. When a new RT task makes a request for QoS guarantess to DSRT, DSRT creates a new instance of this data structure (called *srt_task_struct*) containing information about the state of the particular RT task. This task is also cross-referenced with the *task_struct* structure defined by the Linux scheduler to ensure proper communication between the Linux scheduler and DSRT. More precisely, the data structure contains a pointer to the associated *task_struct* structure, necessary information about the state of the RT task in the DSRT scheduler (running, sleeping, best-effort mode)[8], the period, the cycle demand requested, the number of deadlines missed, the number of periods in which the task tried to overrun and the statistical CPU usage in cycles.

Conceptually, DSRT implements 3 *runqueues* that allow the EDF scheduler and the overrun timer to schedule the tasks. The first runqueue is for the RT task process currently ready to run. The second one is for the RT task processes that are running in best effort mode because they overrun. The last one is for the RT task processes that are awaiting for the beginning of the next period. We implement these runqueues as a single list of processes sorted by the EDF policy. We use the information stored in the *srt_task_struct* about the state of the task to differentiate among different runqueues. We avoid implementing more runqueues due to unnecessary kernel overhead. Also, even though the current DSRT implementation takes $O(n)$ complexity (n is the number of RT tasks) due to the linked list operations, the computation complexity can be brought down to $O(\log(n))$ by using a heap data structure.

To further minimize the number of changes required to the Linux scheduler, the DSRT scheduler does not load or schedule the RT tasks directly, instead it relies on the Linux scheduler. The DSRT simply rises the priority of the running RT tasks to the highest RT priority available on the system, and requests a reschedule to the Linux kernel. This triggers a context switch and forces the Linux scheduler to pick the task that DSRT wants to be scheduled next. To preempt a running RT task to best effort mode when the overrun timer expires, it simply suffices to lower RT task's priority in the Linux scheduler to normal and rise the priority of another RT task. When all the RT tasks have completed the running job, they yield the CPU by invoking the *sched()* function. Upon the

[8] Note that a RT task A_{ij}^{CPU} can also become BE task if it violates its assigned deadline D_{ij}^{CPU} (see discussion in Section 6)

call, the Linux scheduler will take care of scheduling all the BE tasks including iDSRT aware and non-iDSRT aware tasks. The DSRT scheduler remains idle until one of the RT tasks begins a new period. This approach makes the implementation simple and ensure maximal compatibility with non-iDSRT aware tasks.

8.2 iEDF Implementation

Essentially, iEDF is a queuing discipline in Linux. It communicates with the Local Coordinator via the */proc/* file system. This interface includes *create/modify/delete* a local (shadow) task A_{ij}^{Net} and a $SYNC$ signal with the Local Coordinator. iEDF maintains the information of those tasks A_{ij}^{Net} by a double linked list data structure. Each shadow task in $iEDF$ is implemented as a kernel timer that simulates the same behavior as the corresponding task. It is important that even though iEDF may have many timers for shadow tasks, Linux Kernel 2.6 implements these timers as a single high resolution kernel timer to reduce the overhead.

When loaded, iEDF hooks into the queue of the specified network device and replaces the $enqueue()$, $dequeue()$, $requeue()$, $drop()$ functions of the queue with its own functions. The two most important functions are $enqueue()$ and $dequeue()$ function. $enqueue()$ function is invoked when an application requests to send a packet. iEDF captures this packet and extracts the identifier of the application which has sent the packet. If the the application identifier maps to an unregistered identifier, the task is treated as best-effort and the packet is put into the default BE queue. If the application has registered with iEDF, iEDF will put the packet into the corresponding queue of the RT task. Whenever a task is supposed to transmit according to the EDF policy, iEDF grasps a packet from the corresponding task's queue and sends the packet to the MAC. When $dequeue()$ function is called, iEDF will prioritize any RT task available to send. If there is no RT task available, it will grasp a packet in the default queue and send to the MAC. This will ensure the work-conserving semantic of the EDF algorithm. In other words, this is how BE packets are treated. They only get transmitted (in FIFO manner) only when there is no RT packet in the queues of all clients.

iEDF maintains a FIFO queue for packets of each application. Each entry in a FIFO queue is a pointer to *sk_buff* kernel data structure of a real packet. Thus, iEDF works on pointers to packets to avoid any extra data copy overhead. In addition, iEDF maintains a bitmap representing applications that have packets in the FIFO queues (i.e. one bit per application). Whenever a packet of a RT application enters the queue, the corresponding bit is set to 1. This bit will be set to 0 when the last packet leaves the corresponding queue. To look up applications with non-empty FIFO queue, iEDF uses *__ffs()* operator on the bitmap to efficiently search for the 1-bit. Similar to DSRT, the complexity of iEDF scheduling is $O(n)$ (n is the number of RT applications) due to the complexity of maintaining linked lists for the applications' information. This complexity can be brought down to $O(\log(n))$ by using a heap data structure.

9 Evaluation

9.1 Experiment setup

We evaluate our system in a test-bed of 7 laptops. The configuration of laptops is IBM T60 Dual Core 1.66Ghz with 802.11a/b/g Atheros-based wireless card. We disable one core to emphasize the impact of DSRT on CPU scheduling. All laptops run Linux kernel version 2.6.16 with high-resolution timer patch. We setup the laptops as shown in Figure 9. There is one laptop acting as a gateway. The rest of the laptops are clients. To ensure that the gateway is more powerful than the clients, we set the CPU frequency of the server to the highest one (1.66 Ghz) and the CPU frequency of the clients to 1Ghz. All of laptops operate on 802.11a mode and are placed such that they are within the transmission range of each other. The network also operates on the channel that has least interference to minimize external effects.

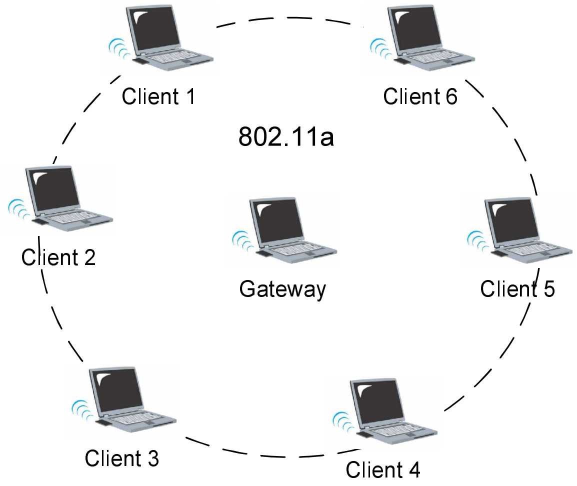

Fig. 9. Testbed setup

9.2 Scenarios

The RT application in the experiment is a regular periodic task creating/reading and sending a packet every 30ms. This is a typical RT task and time requirement for IED devices measurements (e.g. Phasor Measurement Unit devices) as specified in [7]. Each packet encapsulating a PMU measurement with the RTP-like header [5] has the size of 128 bytes. The RTP-like header contains the sequence number for calculating packet losses and time-stamp for clock synchronization and delay calculation.

Other parameters of RT applications such as computation time, network transmission time, subdeadlines are measured and assigned by iDSRT. To scale up the experiments, each client may have more than one RT application. Specifically, we keep adding the RT applications in the system until the admission control fails. Note that for any number of RT applications in the system, these RT applications are distributed equally to clients. For example, if there are 10 RT applications in the system, each client has $\lfloor 10/6 \rfloor = 1$ application and the remaining four RT applications are assigned to any four clients.

Our goal is to to make sure that the system can provide real-time guarantees even there are competing BE applications. On each client we run a very CPU-intensive BE application to compete for the CPU resource. Furthermore, we setup three BE TCP flows from three clients to the server. These three TCP flows will always try to send as much as they can when getting a chance. We use *iperf* [4] as our TCP BE applications.

9.3 Evaluation metrics

We compare our iDSRT system against three other systems. The first one, named as "BestEffort" is the combination of commodity Linux and 802.11 MAC. The second one, named as "DSRT only", is the system with DSRT enabled and iEDF disabled and the last one, named as "iEDF only" has iEDF enabled only and DSRT disabled. The metrics we use are 1) RT end-to-end delay from a client N_i to the gateway S, and 2) the percentage of packet losses of RT applications A_{ij} and 3) the percentage of missing deadlines of RT applications A_{ij}.

All measurements above are done at the gateway S. In every experiment, each application sends 1000 samples, which takes $30ms \times 1000 = 30s$ to finish. The end-to-end delay is measured as the time difference from the packet sent at the client to the time that packet received at the server[9]. The percentage of packet losses are measured by counting the missing sequence numbers. Similarly,

[9] The clocks of these clients are synchronized on every wireless transmission

the percentage of missing deadlines is measured as the number of packets that are received later than the deadline at the gateway. Also, in each scenario, experiments are repeated 5 times to get the average measurements.

9.4 Experiment Results

End-to-End delay: Figure 10 shows the end-to-end delay four different systems. The x-axis represents the total number of applications. The y-axis shows the average end-to-end delay in *ms*. In general, only iDSRT can guarantee the deadlines while the other systems cannot. BE system cannot handle the RT applications well because it does not prevent CPU-intensive application and TCP flows from exhausting CPU and network resources. This makes sense because BE system is designed for general purpose, not for real-time purpose.

DSRT-only system prioritizes RT processes and schedules them according to their deadlines. The CPU resource for RT processes is "reserved", i.e. BE processes cannot compete for that reserved resource. That is the reason why DSRT-only system performs better than the BE system. However, because DSRT-only system can only provide RT guarantee on the CPU resource and it lacks of the RT network scheduler, the end-to-end delay cannot be guaranteed.

iEDF-only system performs worst due to two reasons. The first one is the lack of support from the RT CPU scheduler (i.e. DSRT). The BE CPU scheduler (i.e. Linux scheduler) is done in a round-robin fashion and is not aware of application deadlines. Consequently, packets arrive to iEDF in an aperiodic fashion, which may be earlier or later than the slots iEDF reserves for network transmission. If a packet of a RT task arrives to iEDF later than the slot reserved for its network part, it can only be transmitted until the next reserved time slot releases. This causes cascading missing deadlines of a RT task and can only be fixed with a coordination from the CPU scheduler. This explains why it performs worse than the BE system. The other reason is the shared nature of wireless medium among clients which makes consumed network resource grow quickly as the number of RT applications increase. That explains why iEDF does not scale as good as DSRT.

Lastly, iDSRT with the integration of DSRT, iEDF and the iCoord coordination protocol performs the best. This result validates our design idea in which node scheduler, packet scheduler and task scheduler have to coordinate very well. Missing any components will not be sufficient to provide soft real-time guarantees.

It is also important to emphasize that iDSRT needs a good profiling and an admission control. In our scenario, iDSRT cannot accept more than 13 RT applications due to the admission control. We did run more than 13 applications and the results basically show that, not admitted RT tasks perform much worse than in BE system because most resources in iDSRT are reserved for admitted RT tasks. This, again, underscores the importance of QoS guarantees: once RT tasks are accepted, they will receive what promised by the system.

Missing deadlines: Figure 11 shows the number of missing deadlines of the four systems under various total number of applications. The x-axis shows the total number of applications and y-axis is the percentage of missing deadlines.

The general trend in this figure is similar to the previous one. BE system and DSRT-only system have similar percentages of missing deadlines (30% to 40%). In these systems, the main cause for missing deadline is the lack of network scheduling. iEDF-only performs worst as expected due to the lack of support from CPU scheduling and higher resource-consuming rate of each application. iDSRT performs best and has only around 15% missing deadlines.

This figure also shows the nature of "soft" RT guarantees. The reasons for missing deadlines, even in the case of iDSRT, are preemptions due to hardware interrupts and non-preemptive nature of network scheduling (i.e. iEDF cannot preempt a packet being sent). However, iDSRT with a low average end-to-end delay (around 15ms) and a reasonable missing deadlines (around 15%) is still the best candidate to achieve soft RT guarantees.

To further support that iDSRT is the best candidate, we show the maximum delay of each systems in Table 3. This table essentially shows the worst end-to-end delay of each system in our experiments.

It is clearly shown that iDSRT has the smallest worst end-to-end delay with the deviation of 5ms. In other words, iDSRT misses 15% of deadlines but the deviation is *bounded* in 5ms.

#Apps	Best Effort	DSRT-only	iEDF-only	iDSRT
6	90.06	90.54	51.58	34.17
7	93.75	88.40	69.90	35.38
8	96.88	92.34	82.82	35.59
9	482.05	101.30	102.43	33.78
10	508.31	109.15	138.21	33.59
11	505.76	97.77	154.13	34.17
12	516.69	128.25	164.42	34.63
13	558.37	136.11	169.35	35.25

Table 3. Maximum EED (ms)

Packet Losses Figure 12 shows the packet losses of four systems. As shown clearly in the figure, all systems have very low packet loss rate (less than 0.1%). iEDF in this case shows the advantage of very low (almost no) packet losses due to the distributed scheduling mechanism. iDSRT, again, inherits the advantage of both DSRT and iEDF to achieve almost no packet loss.

10 Related Work

There has been large amount of research work that address individual, part of the end-to-end RT problem and can be classified into three categories: real-time operating system, real-time wireless network and end-to-end delay guarantee. The first category addressing real-time operating system has been extensively explored. Typical work in this category includes hard real-time solutions such as LynxOS [2], RTLinux [9]; firm real-time solutions such as Rialto [24], SMART[26], KURT [30] and soft real-time solutions such as DSRT [22], GraceOS [34]. An important observation is that the harder real-time guarantee is, the more support from special hardware or modification of operating systems are required and thus the less accessability to wide-range of operating system services. Furthermore, these works only address the real-time aspect of operating systems, which is a part of the end-to-end delay guarantee.

The second category addressing real-time wireless network has been also well explored. A typical work is the 802.11e standard [8]. Although 802.11e becomes the standard and commercially available, its prioritization mechanism does not work well when there are multiple flows with the same priority. In fact, it even increases more collisions due to its aggressive medium access parameters such as smaller CW_{min} and CW_{max}. Besides 802.11e, there are plethora of work involving with MAC design such as Black Burst contention [32][33][28], RI-EDF [16] or Wireless Token Ring [17]. Even though these schemes can work well, they require MAC modification and thus make it less compatible and more expensive to the work that exploits standard services and available hardware. We believe that with the wide-spread availability of 802.11-based hardware, it is much cheaper and more applicable to have a solution working on top of and independent of 802.11 MAC. Several works sharing this view includes Overlay MAC [27] and middleware-based control [21] [20]. Unfortunately, these works only address the real-time aspect of the wireless networks and ignore the real-time aspect of the operating systems. Thus, again they cannot provide the full end-to-end delay guarantee.

In the third category, essentially, previous work has shown the need for integrating the task scheduler and the network scheduler referred to as multi-resource coordination/reservation and scheduling problem [18][19][31][29]. In [29][25], the approach is to allocate the resources such that the end-to-end delay can be guaranteed while optimizing the general resource utilization. Xu et al. [31] tries to provide best end-to-end QoS level for an application under the constraints of resource availability in wired networks. In [18][19] end-to-end delay is achieved by assigning deadlines for each resource

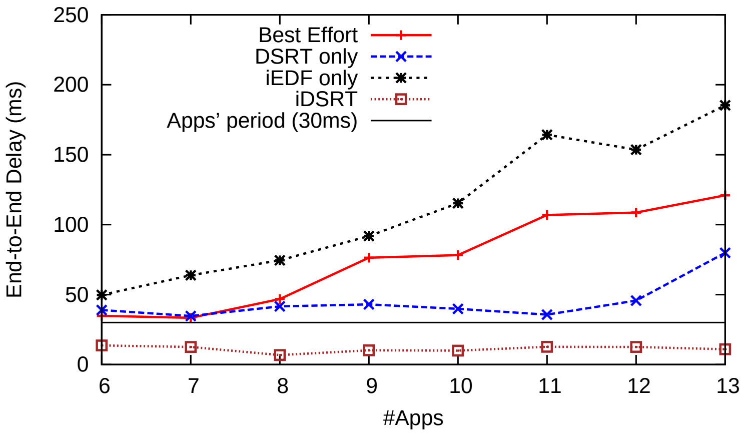

Fig. 10. End-to-End Delay

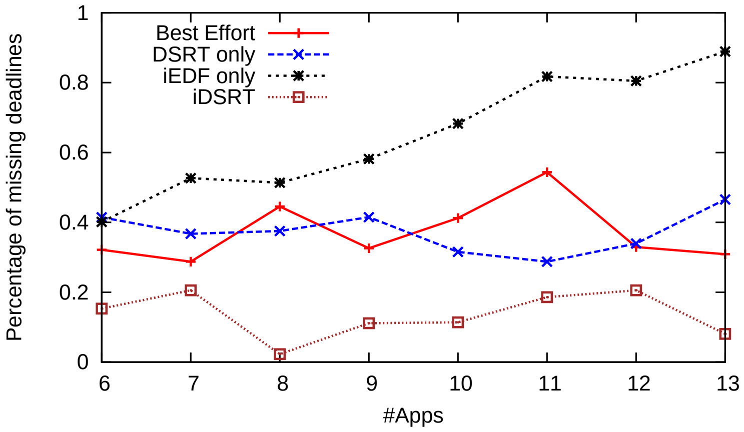

Fig. 11. Missing deadlines

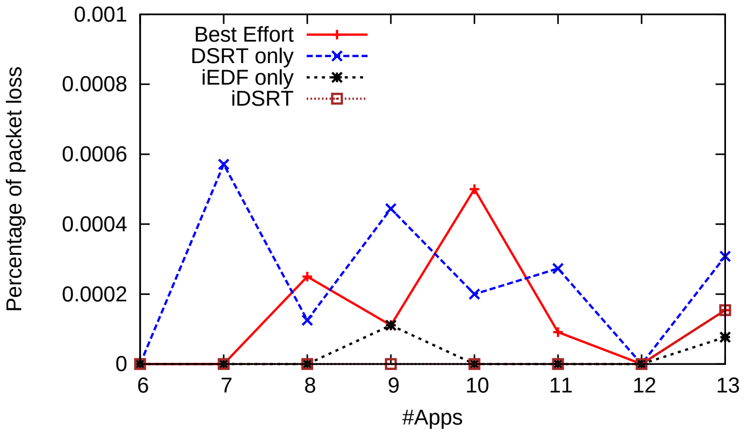

Fig. 12. Packet Loss

such that the number of future applications admitted is maximized. However, the works consider only the wired networks and are not applicable to the wireless networks.

11 Conclusions

We have shown an integrated soft real-time scheduling framework, i.e. multi-resource allocation and scheduling for periodic soft-real-time tasks in wireless LAN environment. This is the first integrated system that considers both scheduling and coordination of three important entities in WLAN: the RT tasks, the RT packets and the nodes that share the wireless medium. The result of iDSRT clearly show that augmented Linux and 802.11 WLAN technologies are feasible for critical infrastructures such as PowerGrid SCADA systems and can yield delay and loss guarantees currently only achievable over the wired network with modified general purpose kernels. We believe that iDSRT allows an easy deployment of general purpose hardware and software in PowerGrid substation, while preserving a major requirement of the real-time guarantees.

References

1. General electric wireless SCADA/Telemetry networking, http://www.microwavedata.com/applications/scada/.
2. LynxOS - Hard real-time OS features and capabilities.
3. SEL-3022 wireless encrypting transceiver, http://www.selinc.com/sel-3022.htm.
4. The TCP/UDP bandwidth measurement tool, http://dast.nlanr.net/projects/iperf.
5. RTP: A transport protocol for real-time applications, RFC 3550, July 2003.
6. IEEE P1777/D1: Draft recommended practice for using wireless data communications in power system operations, February 2007.
7. IEEE Standard 1646: Communication delivery time performance requirements for electric power substation automation.
8. IEEE standard 802.11e, http://standards.ieee.org/getieee802/802.11.html, September 2004.
9. Ayers, and Yodaiken, B. V. Introducing real-time linux. *Linux Journal 1997*, 34es (1997), 5.
10. Baruah, S., Rosier, L., and Howell, R. R. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. In *Journal of Real-time Systems* (1990).
11. Bertsekas, D. P. *Nonlinear Programming.* Athena Scientific, 1999.
12. Bini, E., and Buttazzo, G. The space of EDF feasible deadlines. In *19th Euromicro Conference on Real-Time Systems (ECRTS)* (2007).
13. Boyd, S., and Vandenberghe, L. *Convex Optimization.* Cambridge University Press, 2004.
14. Buttazzo, G. *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, 1997.
15. Caccamo, M., Zhang, L. Y., Sha, L., and Buttazzo, G. An implicit prioritized access protocol for wireless sensor networks. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)* (2002).
16. Crenshaw, T. L., Hoke, S., Tirumala, A., and Caccamo, M. Robust implicit EDF: A wireless MAC protocol for collaborative real-time systems. In *Transaction on Embedded Computing System* (2007).
17. Ergen, M., Duke Lee, R. S., and Varaiya, P. WTRP: Wireless token ring protocol. In *IEEE Transaction on Vehicular Technology* (2004).
18. Gopalan, K., and cker Chiueh, T. Multi-resource allocation and scheduling for periodic soft real-time applications. In *Proceedings of ACM/SPIE Multimedia Computing and Networking* (2002).
19. Gopalan, K., and Kang, K.-D. Coordinated allocation and scheduling of multiple resources in real-time operating systems. In *Proceedings of Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)* (2007).
20. He, W., and Nahrstedt, K. Impact of upper layer adaptation on end-to-end delay management in wireless ad hoc networks. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS),* (2006).
21. He, W., Nguyen, H., and Nahrstedt, K. Experimental validation of middleware-based QoS control in 802.11 wireless networks. In *3rd International Conference on Broadband Communications, Netwoks, and Systems (BROADNETs)* (2006).
22. hua Chu, H. *CPU Service Classes: A Soft Real Time Framework for Multimedia Applications.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

23. Jeffay, K., Stanat, D., , and Martel, C. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS)* (1991).
24. Jones, M., Alessandro, J., Paul, F., Leach, J., RoOu, D., and RoOu, M. An overview of the rialto realtime architecture, 1996.
25. Nahrstedt, K., hua Chu, H., and Narayan, S. QoS-aware resource management for distributed multimedia applications. *Journal on High-Speed Networking, Special Issue on Multimedia Networking Vol. 8* (1998), pp. 227–255.
26. Nieh, J., and Lam, M. S. The design of SMART: A scheduler for multimedia applications. Tech. Rep. CSL-TR-96-697, 1996.
27. Rao, A., and Stoica, I. An overlay MAC layer for 802.11 networks. In *3rd International Conference on Mobile Systems, Applications, and Services* (2005).
28. Sobrinho, J. L., and Krishnakumar, A. S. Quality-of-service in ad hoc carrier sense multiple access networks. In *IEEE Journal on Selected Areas in Communications* (1999).
29. Sourav Ghosh, Ragunathan Rajkumar, J. H., and Lehoczky, J. Integrated resource management and scheduling with multi-resource constraints. *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)* (2004).
30. Srinivasan, B., Pather, S., Hill, R., Ansari, F., and Niehaus, D. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium (RTAS)* (1998).
31. Xu, D., Nahrstedt, K., Viswanathan, A., and Wichadakul, D. Qos and contention-aware multi-resource reservation. *IEEE International Symposium on High Performance Distributed Computing (HDPC)* (2000).
32. Yang, Y., and Kravets, R. Achieving delay guarantees in ad hoc networks through dynamic contention window adaptation. In *IEEE Conference on Computer Communication (INFOCOM)* (2006).
33. Yang, Y., Wang, J., and Kravets, R. Distributed optimal contention window control for elastic traffic in wireless LANs. In *IEEE Conference on Computer Communication (INFOCOM)* (2005).
34. Yuan, W. *GRACE-OS: An Energy-Efficient Mobile Multimedia Operating System*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004.