Brigham Young University
BYU ScholarsArchive

Theses and Dissertations

2006-11-17

A Performance Evaluation of Dynamic Transport Switching for Multi-Transport Devices

Lei Wang
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

 Part of the Computer Sciences Commons

BYU ScholarsArchive Citation

Wang, Lei, "A Performance Evaluation of Dynamic Transport Switching for Multi-Transport Devices" (2006). *Theses and Dissertations*. 824.
https://scholarsarchive.byu.edu/etd/824

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

A PERFORMANCE EVALUATION OF DYNAMIC TRANSPORT SWITCHING FOR MULTI-TRANSPORT DEVICES

by

Lei Wang

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

December 2006

Copyright © 2006 Lei Wang

All Rights Reserved

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Lei Wang

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Charles D. Knutson, Chair
Date	Daniel M.A. Zappala
Date	Yiu-Kai Dennis Ng

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Lei Wang in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Charles D. Knutson
Chair, Graduate Committee

Accepted for the Department

Parris K. Egbert
Graduate Coordinator

Accepted for the College

Thomas W. Sederberg
Associate Dean,
College of Physical and Mathematical Sciences

ABSTRACT

A PERFORMANCE EVALUATION OF DYNAMIC TRANSPORT SWITCHING FOR MULTI-TRANSPORT DEVICES

Lei Wang

Department of Computer Science

Master of Science

Multi-transport devices are becoming more common, but sophisticated software is needed to fully realize the advantages of these devices. In this paper, we examine the performance of dynamic transport switching, which selects the best available transport for communication between two devices. We simulate transport switching within the Quality of Transport (QoT) architecture and show that it can effectively mitigate the effects of congestion and interference for connections between two multi-transport devices. We then evaluate dynamic transport switching overhead to characterize its effect on application throughput. Based on these insights, we identify several limitations of the QoT architecture and present solutions to improve performance.

ACKNOWLEDGMENTS

I would like to thank my wonderful wife, Qiuyi, for her understanding, great support, and especially the fantastic cooking through this long process.

To my advisor, Dr. Knutson, thank you for the patient and insightful help on my research. It's hard to believe how much I have improved under your instruction over the past three years. Thank you for loving me like your own child. I feel so lucky to have you as my advisor.

To Dr. Zappala, thank you for pointing out an efficient and effective direction on my research when I feel so frustrated. Thank you for your precious advice on my thesis.

Finally, I thank Manoj for helping me through the tedious and desperate learning curve of NS-2. It feels fantastic when I can do seriously "bad" things to NS-2. It seems that all those painful efforts are finally paid off.

Contents

List of Figures

Chapter 1

Introduction

Mobile devices equipped with multiple wireless networking interfaces are becoming increasingly common. A mobile device, such as a Personal Digital Assistant (PDA), may support cellular network access as well as ad hoc connections with other mobile devices through built-in Bluetooth and infrared ports. With such intra-device multi-transport[1] heterogeneity, devices can potentially receive and transmit signals via several wireless networking interfaces.

Multi-transport devices can be used to improve user experience in many ways. One example could be facilitating photo transfer between a user's cell phone and laptop computer. Both the cell phone and laptop computer support networking methods via the Bluetooth and WiFi interfaces. The user may initiate the photo transfer over the WiFi interface. If the WiFi link becomes severely interfered or congested, the cell phone automatically and intelligently switches the photo transfer to Bluetooth to improve transmission quality.

Multi-transport devices can also be utilized in mesh networking. Ad hoc networks have historically been comprised of devices that support a single transport mechanism. Building an ad hoc network using multi-transport devices can significantly expand its capability. As an example, Fig. 1.1 shows a mobile ad hoc network composed of multi-transport nodes. In this example, two nodes should be able to communicate through whatever transports are available. Each hop on a path may potentially utilize a different transport, with

[1] By "transport" we refer broadly to traditional stovepipe communication architectures interfaced primarily via the transport layer of the protocol stack. Hence, when we use the term "transport," we refer to all layers from the transport layer to the physical layer inclusive. As an example, we would refer to Bluetooth and WiFi as separate "transports."

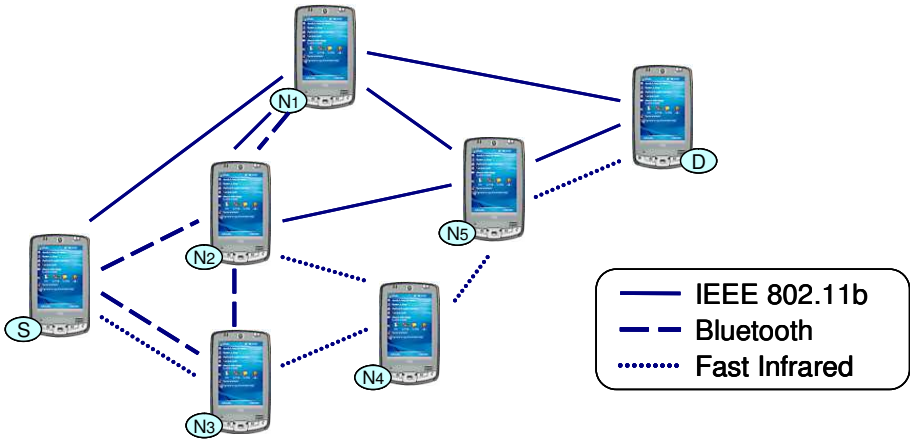

Figure 1.1: Mobile Ad Hoc Network composed of multi-transport nodes

the source node dynamically choosing between available paths based on observable performance. As illustrated in Fig. 1.1, node S is communicating with node D. For this communication, there are potential paths, with one possibility of using Bluetooth from S to N_2, WiFi from N_2 to N_5, and Fast Infrared (FIR) from N_5 to D.

Whether for point-to-point or mesh networking, a needed feature is the ability to dynamically choose the best available transport at each hop. A particular hop on a path may decide to switch transports for any of the following purposes:

- *Preserve Connectivity.* Nodes can maintain robust connectivity by utilizing dynamic transport switching. Assume S communicates with N_3 via FIR. At a later time, N_3 moves away from S and the distance between them is beyond the range of FIR. Node S can seamlessly switch the data communication from FIR to Bluetooth to maintain the connection with N_3.

- *Improve Link Quality.* Nodes can provide better communication link quality by means of dynamic transport switching. For example, node N_5 and node D both use WiFi and FIR. Assume N_5 communicates with D via WiFi. This interface could become congested by ambient IEEE 802.11b traffic or interference from other wireless sources operating in the 2.4 GHz ISM band. Node N_5 can avoid such congestion

and interference by switching the active transport from WiFi to FIR.

- *Conserve Power.* Nodes can achieve longer battery life by using interfaces that consume less power. Assume node S switches the active transport from FIR to Bluetooth to maintain a connection with node N_3. Should N_3 move within the range of FIR, S could switch the data communication back to FIR to conserve power.

Despite the benefits demonstrated above, dynamic transport switching mechanism may also incur overhead to data transfer. For multi-transport devices, the networking interface that is not being used is normally powered off to preserve power. To determine the availability of remote devices via a particular transport, a device must power on the interface periodically and query for potential connectivity with remote devices. Both of these operations can interfere with data transmission. To lay a foundation for further transport switching protocol design, we need to assess the impact of this overhead on application performance.

In this paper, we use simulations to study the performance of dynamic transport switching. We first demonstrate that dynamic transport switching can effectively mitigate the negative effects of congestion and interference for single-hop connections, which form the basis for multi-hop connections in ad hoc networks. We then evaluate the potential overhead of dynamic transport switching for point-to-point communication. The overhead is evaluated first in scenarios of only a single pair of nodes and then when multiple pair of nodes are communicating. We also address two performance problems that originate from data buffering within the QoT architecture. We identify their impact on data throughput and present solutions to improve performance.

Chapter 2

Related Work

Prior research on dynamic transport switching in heterogeneous wireless environments has typically taken one of two forms: Infrastructure-based or point-to-point ad hoc.

2.1 Infrastructure-based transport switching

Infrastructure-based transport switching involves passively listening to beacon messages from a wireless access point to ascertain the presence of a wireless network. This is only applicable to infrastructure-based wireless networking, and is not suitable for either peer-to-peer or ad hoc communication.

The BARWAN project at the University of California at Berkeley proposes the concept of a vertical handoff system that allows users to roam between cells in wireless overlay networks [1]. In a vertical handoff system, all network interfaces are turned off by default with the exception of the overlay immediately below the current overlay. This overlay wakes up periodically to listen to beacons on the lower interface for a short time. The BARWAN mechanism is based on the assumption that the reason for traffic switching is to simply roam into or out of the service coverage of wireless networks. This assumption may not hold in a more flexible usage scenario where traffic switching may also happen in order to preserve power or improve link quality.

Research at Georgia Institute of Technology uses a similar concept of the vertical handoff and proposes a TCP scheme for a seamless vertical handoff between WLAN and 3G cellular networks [2]. In their performance evaluation, the proposed scheme avoids

packet loss during the handoff and reaches a stable condition rapidly.

The MosquitoNet project at Stanford University implements a mobile IP system that supports seamless switching between different networks and communication devices [3]. The measurements of their implementation show that the inherent overhead to switch networks is insignificant compared to the time required to bring up a new communication device.

The capability of dynamic transport switching has also drawn attention from industry researchers. The WLAN-GPRS integration project at Motorola aims to provide users ubiquitous data services and very high data rates in hotspot locations [4]. They discuss the general aspects of integrating WLANs and cellular data networks and examine the generic internetworking architectures. The IOTA project at Lucent focuses on providing users seamless roaming across 802.11 and 3G networks [5].

2.2 Ad hoc transport switching

In ad hoc transport switching, a node must actively probe the supported wireless networking interfaces to determine their availability. The following projects present dynamic transport switching mechanisms that may be suitable for either peer-to-peer or ad hoc communication.

The WiOptiMo project proposes an application layer solution to facilitate seamless handover between wireless networks, such as Bluetooth, 802.11x local area networks and 3G cellular networks [6]. This solution is designed for pairs of communicating devices and can perform either infrastructure-based or ad hoc transport switching. In their performance evaluation, the authors show that the throughput is not largely affected by the wireless network handover; in the worst case the proposed switching mechanism reduces the throughput by less than 2%. However, the overhead due to periodic transport availability queries are not evaluated. Furthermore, the transports used in their experiments, GPRS and WiFi, do not share an overlapping frequency band, so they do not interfere with each other. Transports with overlapping frequencies may severely interfere with each other, and thereby generate significant overhead to data communication.

IEEE 802.21 is an emerging standard for Media Independent Handover (MIH) services [7] [8]. This standard proposes a link layer solution to optimize handovers between heterogeneous networking technologies. It supports algorithms enabling seamless handover between networks of IEEE 802 series networks, such as WiFi, Bluetooth and WiMAX, as well as between IEEE 802 networks and non-802 networks, such as cellular and wired networks. In IEEE 802.21, a mobile terminal determines the presence of a wireless network through reception of either a beacon or a response to a probe. No work on performance evaluation of IEEE 802.21 has yet been published.

The Quality of Transport (QoT) project aims to facilitate ad hoc data exchange between two mobile devices by means of intelligent, dynamic transport switching [9]. This project also utilizes the characteristic of devices with multiple networking interfaces to substantially reduce the cost of Bluetooth device discovery and connection establishment phases [10], and to maximize data throughput through inverse multiplexing [11].

Although all three projects described above are potential dynamic transport switching mechanisms that may be used for peer-to-peer or ad hoc communication, we use QoT as an example dynamic transport switching mechanism to conduct our research. The WiOptiMo project assumes common support to TCP/IP protocol stack, and hence narrows its usage models. IEEE 802.21 requires significant modifications to current protocols at the data link layer, and the technical efforts that are required to accomplish this are not clear at this stage.

Chapter 3

Heterogeneous Ad Hoc Networking with QoT

QoT is a protocol layer residing between the session layer and the transport layer in the OSI reference model. As illustrated in Figure 3.1, it works as a proxy layer between applications and underlying reliable connection mechanisms. QoT bridges the upper Transport Proxy Module (TPM) and the lower Transport Abstraction Module (TAM), which are specific to each supported transport [9]. The TPM appears to a session layer as if it were an interface to a specific transport, even though the transport that actually transfers user data may change during the communication. The TAM interacts with the transport layer as if it were an arbitrary (but indeterminate) session protocol.

QoT is designed to exploit intra-device heterogeneity to optimize communication quality by means of dynamic transport switching. Figure 3.2 illustrates a QoT-enabled data exchange between two devices using session protocol S_1. The two devices each support three transports, two of them common (T_2 and T_3). In this figure, the highest quality link is provided by T_3, so QoT routes the traffic of session protocol S_1 via transport protocol T_3 (in dashed lines). Should link conditions change such that T_2 provides a more desirable link, QoT would switch the underlying transport to T_2 (in solid lines) without affecting the data exchange.

QoT-enabled multi-transport nodes conduct periodic queries to determine the status of networking interfaces in order to facilitate dynamic transport switching. This is because pairs of communicating nodes cannot rely on beacons from infrastructure networks to ascertain each other's presence.

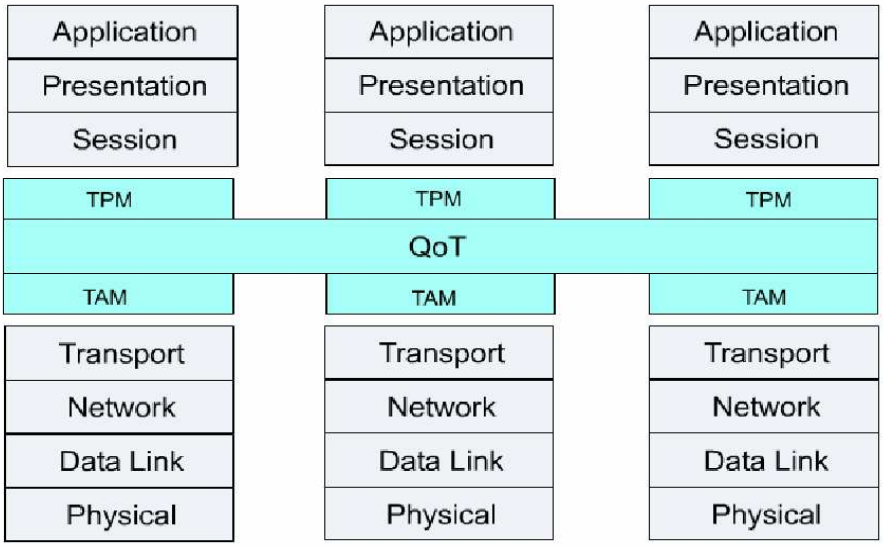

Figure 3.1: QoT in OSI reference protocol stack

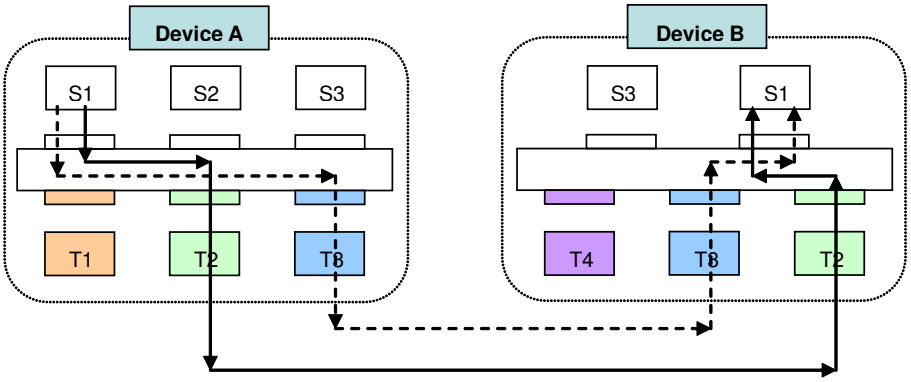

Figure 3.2: Data exchange using QoT

When communicating with a peer, the QoT layer on the sending node buffers packets from the session layer and assigns sequence numbers to them before sending them down to the active transport layer protocol. When switching transports, the QoT layer on the sending node must know where to resume the data communication when performing transport switch. Once the buffer is full, the QoT layer sends a data synchronization message to the corresponding QoT layer on the receiving node, which responds with the sequence number of the packet most recently received. When the buffer is full, QoT cannot handle further packets from the session layer. Upon receiving the synchronization response, the QoT layer releases packets that are acknowledged in the buffer and continues receiving packets from the session layer.

Chapter 4

Transport Switching in NS-2

In order to evaluate the impact of dynamic transport switching on point-to-point data communication, we made three significant modifications to Network Simulator 2 (ns-2). First, we implemented intra-device heterogeneity that permits individual nodes to support multiple transport mechanisms. Second, we implemented a dynamic transport switching mechanism, modeled after QoT, that permits data communication between two nodes to seamlessly continue during a transport switch. Finally, we made some modifications to the ns-2 implementation of TCP so that it sends actual packets.

4.1 Heterogeneous nodes

As Figure 4.1 shows, we facilitate multi-transport nodes in ns-2 by subsuming ordinary homogeneous nodes (such as WiFi, Bluetooth and ZigBee) within a newly-defined heterogeneous node structure. Compared to a homogeneous node, a heterogeneous node is a *"virtual"* node in the sense of not possessing traditional protocol stack layers such as channel layer, physical layer, data link layer, routing layer and transport layer. Instead, a heterogeneous node may include one or multiple ns-2 homogeneous nodes in order to maintain the appearance of a single heterogeneous node. Finally, traffic generators, such as File Transfer Protocol (FTP) or Constant Bit Rate (CBR), are associated with a heterogeneous node rather than being linked to a *static* transport agent.

Homogeneous nodes may co-exist with heterogeneous nodes in the same simulation. Heterogeneous nodes can communicate with either heterogeneous or homogeneous

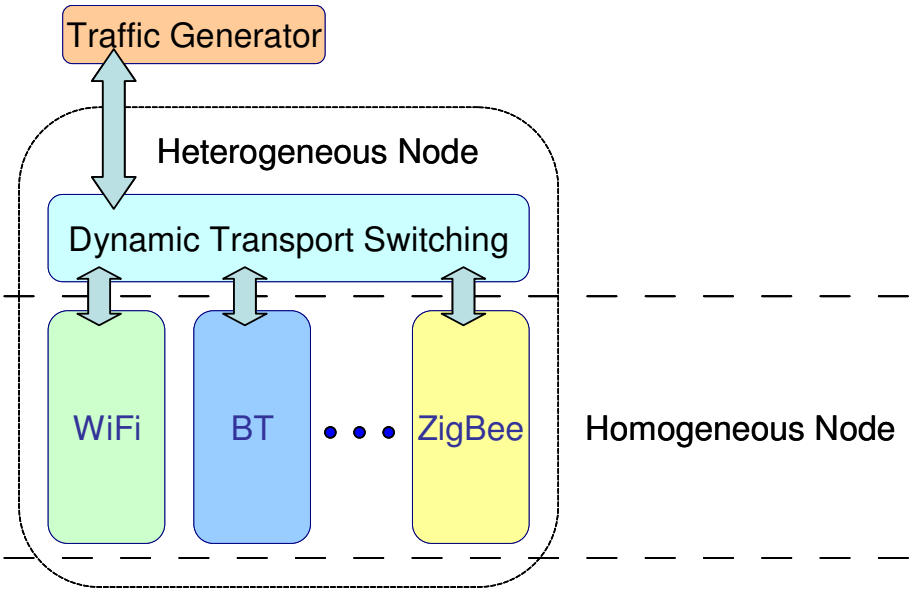

Figure 4.1: Heterogeneous Node in NS-2

nodes, but cannot perform transport switching while communicating with homogeneous nodes.

4.2 Dynamic transport switching

Upon receiving packets from a traffic generator, the transport switching mechanism of a heterogeneous node determines which transport is to be used for data communication. As shown in Figure 4.1, the transports of a heterogeneous node are linked to its transport switching mechanism. Since the transport being used for data communication may change, a heterogeneous node must know the address and port number of the active transport in the communicating node. In the connection establishment phase, two communicating nodes exchange information about the address and port number of their transports. They then negotiate to determine the transport that is to be used for data communication. When sending data packets, the transport switching mechanism of the sending node informs the chosen transport of the address and port number of the corresponding transport on the remote side to which packets are sent.

The link quality may change over time in mobile ad hoc networks. In order to know if a transport is available and has the quality needed for an impending transport switch, heterogeneous nodes conduct periodic transport availability queries on each transport shared with the remote node. These queries gather measurements about link quality, and QoT can then make intelligent decisions concerning the “best" transport on which to transmit packets.

4.3 Modifications to TCP

In order to properly test transport switching, TCP agents on both the sending and receiving sides of a communication must send actual packets. This is because the transport switching mechanisms on both sides must communicate with each other when they establish a connection and conduct transport switching.

Current TCP implementations in ns-2 don’t transmit actual packets, but instead record the size of the packets being transmitted and ignore actual data. We modified the `Agent/TCP/Fulltcp` implementation to be able to transmit packets of transport switching mechanisms in our project.

Current TCP implementations in ns-2 also don’t retransmit actual packets, but rather record the sequence number of the packets that need to be retransmitted. In order to facilitate potential packet retransmissions, we also implemented send buffers in the TCP agent.

Chapter 5

Multi-Transport Heterogeneity and Dynamic Transport Switching

In this section we present simulation results utilizing the dynamic transport switching mechanism of multi-transport nodes to avoid collisions and interference among wireless technologies. All the data are average values from the results of five repeated simulations.

5.1 Avoiding collisions

In high density ad hoc networks, collisions between signals of devices that are communicating with each other via a wireless technology may significantly affect data throughput. Heterogeneous nodes are able to mitigate the potentially negative effects of collisions by intelligently switching data communication to less congested transports.

In our simulation, a pair of heterogeneous nodes, each supporting WiFi and Bluetooth, communicate with each other at a distance of 5 meters. As shown in Fig. 5.1, four additional pairs of homogeneous nodes are created to communicate with each other through WiFi. All the heterogeneous nodes and homogeneous nodes are within WiFi range of each other so all nodes could contend for the shared wireless media when all of them use WiFi to transmit signals. All nodes remain static during the lifetime of simulation.

An FTP traffic flow starts from node 1 to node 2 at second 1, and the connection is established over WiFi. At second 8, four FTP sessions start at the four homogeneous node pairs. The two heterogeneous nodes switch the active transport from WiFi to Bluetooth at second 14. The simulation ends at second 20. The simulation result is shown in Fig. 5.2. Fig. 5.2 represent data throughput over time collected at the QoT layer of node 2.

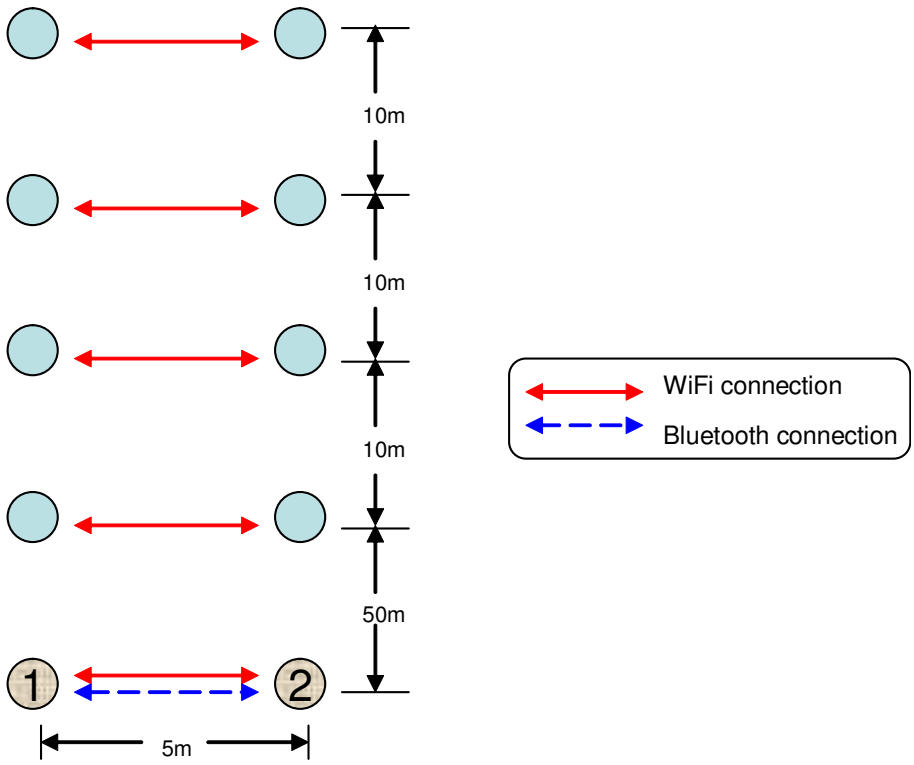

Figure 5.1: Use Transport Switching to Avoid Collisions

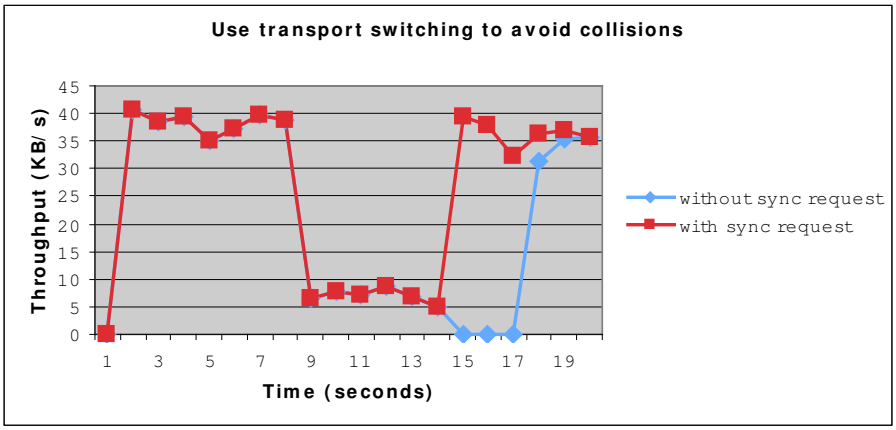

Figure 5.2: Avoiding Collisions

Our simulation results show that transport switching can help a connection achieve greater throughput by switching away from a transport that suffers a high collision rate. Once the homogeneous nodes start data traffic at second 8, WiFi becomes crowded. Collision occurs when more than one node requests to send packets at a same time. The Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) mechanism of the WiFi Media Access Control (MAC) layer backs off for a random time period before trying to send packets again. Such backoffs degrade throughput in our simulation by up to 87.52%.

We expected data throughput to improve after transport switching, but instead it dropped to 0 for three seconds (as shown in the line marked with "*without sync request*"). The reason for this throughput drop lies in the mechanism employed by QoT to perform upgrade transport switching. Recall that QoT buffers packets and must receive a synchronization packet from the receiver to clear its buffer. This causes a problem if the synchronization packet is lost. In our simulation, the QoT layer on node 1 found its output buffer full and requested a data synchronization before the transport switching. Since WiFi was suffering from serious collisions, this synchronization request was also delayed. After about 3 seconds, node 1 received synchronization response from node 2, released the acknowledged packets from its output buffer and resumed data transfer. Hence the data throughput goes up again after second 17. Switching to Bluetooth improves throughput because there are no competing connections on this transport. Note that we implemented the Adaptive Frequency Hopping (AFH) in Bluetooth in ns-2, so that it does not interfere with WiFi in this case.

To fix this problem, we modified the QoT specification so that the QoT layer on the sending node immediately requests data synchronization on the new transport after conducting upgrade transport switching. Since the new transport typically performs better than the previous transport, there is a better chance that this synchronization packet gets through quickly, enabling data communication to resume sooner. In Fig. 5.2, the line marked by "*with sync request*" shows the result with this fix. Data throughput quickly goes up after the transport switching from WiFi to Bluetooth. This demonstrates that dynamic transport switching of heterogeneous nodes can effectively avoid collisions in ad hoc networks.

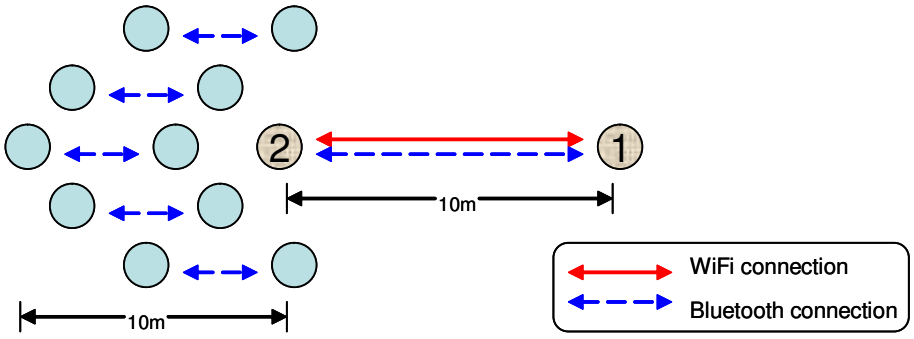

Figure 5.3: Use Transport Switching to Avoid Interference

As observed in Fig. 5.2, the data throughput drops by about 18.5% after the transport switching even with the above fix. This is caused by the overhead of the transport availability query, which will be further discussed in following section.

5.2 Avoiding interference

Interference between disparate wireless technologies, especially in the 2.4 GHz unlicensed ISM band, may negatively affect data throughput. Heterogeneous nodes can avoid interference by dynamically switching to a less noisy transport. In this simulation, we disabled the AFH function of Bluetooth so that there is overlapping on frequency band between WiFi and Bluetooth. By doing so, signals from WiFi and Bluetooth may interfere with each other in our simulation.

As shown in Fig. 5.3, a pair of heterogeneous nodes, node 1 and node 2, communicate with each other. Both of them support WiFi and Bluetooth. We also place five pairs of homogeneous Bluetooth nodes around node 2. Node 2 is within the ranges of homogeneous Bluetooth nodes, while node 1 is not. All nodes remain static during the life time of simulation. In the simulation, the traffic model between node 1 and node 2 is the same as described in the previous section.

Fig. 5.4 shows that switching from WiFi to Bluetooth can avoid interference and increase data throughput. In this simulation, we plot data throughput for the case when

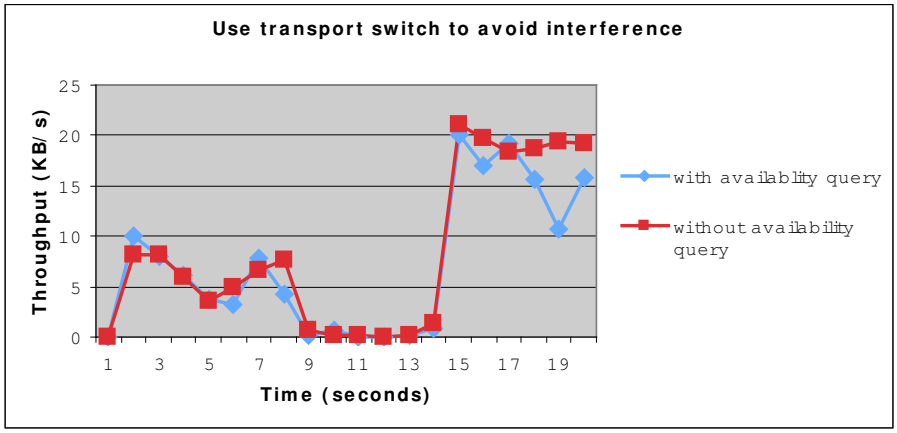

Figure 5.4: Avoiding Interference

QoT conducts periodic transport availability queries, and when it does not. At second 8, five FTP sessions start at the five homogeneous node pairs. Since the active transport that heterogeneous nodes used to transmit data traffic is WiFi, and it is severely interfered by Bluetooth signals, data throughput drops beginning at second 8. At second 14, the two heterogeneous nodes switch the active transport from WiFi to Bluetooth. Bluetooth improves throughput because it performs frequency hopping at a rate of $625\ \mu s$ and hence suffers less from frequency interferences when compared to WiFi.

A major problem with QoT occurs when it conducts transport availability queries, causing QoT to stop transmitting any data for a short period of time. At second 14, the data throughput values collected with transport availability queries are in most cases smaller than those collected without transport availability queries. This is because QoT stops packet transfers when performing a transport availability query until it receives a query response or the query timer expires. QoT queries all the shared transports (WiFi and Bluetooth in this simulation) one after another. This is to avoid potential overlapping during the time when availability query packets are sent, which would cause erroneous query results. If a query response cannot get back quickly, data throughput is negatively affected. The extent to which data throughput might be affected is decided by the ambient wireless environment

of the communicating heterogeneous nodes and the length of the query timeout. In this simulation, we use a query timeout of 500 *ms*, and observe the largest throughput degradation at second 19, where throughput degrades by 43%.

A potential solution for this "*freeze*" problem is to utilize a mechanism that can intelligently separate frequency bands used by the transports within a device. If operating on the same frequencies, transports co-located on a device may interfere severely with each other when transmitting or receiving packets at the same time. One example is the RIA project [12], in which Bluetooth avoids frequencies on which it detects WiFi interference. Device manufacturers could also partition the spectrum among the networking interfaces in the device. However detailed discussion on potential solutions to this problem is not within the scope of this paper.

One interesting aspect of this simulation is that the throughput achieved in Fig. 5.4 is lower than the corresponding values in Fig. 5.2, due to Bluetooth interference on the WiFi transport. The master Bluetooth node (the Bluetooth transport at node 1) continues to poll the slave (the Bluetooth transport at node 2) to ascertain the status of the link. Polling packets and the corresponding response packets may be sent at the same moment that WiFi transmits packets. Such overlapping on the time of packet sending is caused by the way in which a heterogeneous node is created in ns-2. A heterogeneous node is composed of several homogeneous nodes. Protocols of a homogeneous node may generate and send their own packets, such as the Reqest to Send (RTS) packet from the WiFi MAC layer and the POLL packet from the Bluetooth Baseband layer. We schedule the timings that the packets are to be sent at the QoT layer, but not at lower protocol layers. This is because controlling the timings at all protocol layers would introduce too much runtime overhead to ns-2.

Chapter 6

Transport Availability Query Overhead

As has been shown in previous sections, the dynamic transport switching mechanism in heterogeneous nodes generates overhead due to periodic transport availability queries. In this section, we evaluate the impact of transport availability queries in heterogeneous nodes on performance in mobile ad hoc networks.

6.1 Query overhead between two communicating nodes

In this section, we evaluate the overhead of transport availability queries on data transfer when there is only one pair of heterogeneous nodes communicating with each other. This is a common scenario in a user's daily life, such as between his/her cell phone and laptop computer in the office.

We conduct two simulations, one with UDP simulating real-time applications and another with TCP simulating applications that require reliable data transmission. Two heterogeneous nodes, 5 meters apart, transmit packets via WiFi. Both nodes remain static during the simulation, and both support two transports — Bluetooth and WiFi. To observe the overhead of transport availability queries, we vary the query interval from 20 seconds to 0.5 seconds. We then average throughput over a simulation time of 20 seconds at the QoT layer of the receiving node. To prevent the overhead of availability queries being overwhelmed by other interfering factors, we turned on the AFH function of Bluetooth so that it does not interfere with WiFi. Since there are only two nodes in this scenario, there is also no collision between WiFi traffic.

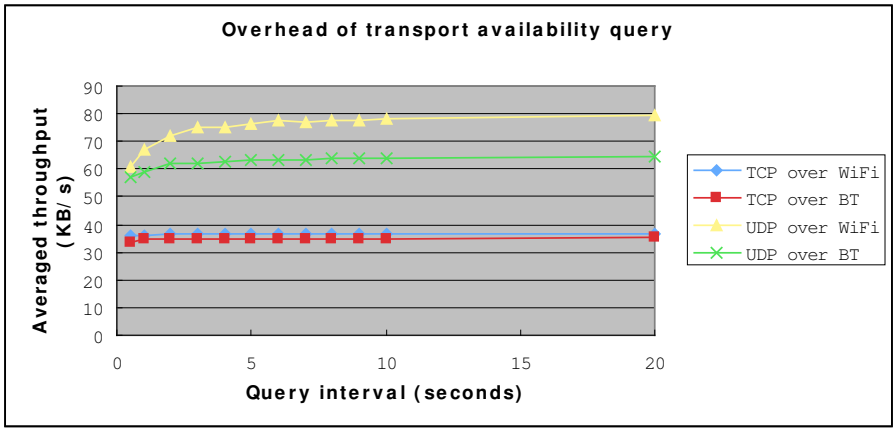

Figure 6.1: Query Overhead Between Multiple Communicating Nodes

Our first UDP simulation shows that transport availability queries do not significantly impact a low-rate UDP flow. We use Constant Bit Rate (CBR) as a traffic generator. In ns-2, CBR generates packets with a specified size at a predefined interval. We set CBR to generate a 64 Byte packet every 100 *ms*. The average throughput does not drop significantly. This is because UDP can send out small packets very quickly, and becomes idle for the rest of the time in the relatively large inter-packet gap. In such a situation, sending transport availability query packets would not delay CBR packet transfer.

Our second UDP simulation shows that transport availability queries can significantly impact a high-rate UDP flow. We set CBR to generate a 512 Byte packet every 3 *ms*, yielding the results shown in Fig. 6.1. With such a work load, UDP keeps busy sending newly generated CBR packets, and transport availability query packets delay CBR packet transmission. When we increase the transport availability query interval from 0.5 seconds to 20 seconds, the average throughput is increased by 23.59% over WiFi and by 11.15% over Bluetooth. When using Bluetooth, the average data throughput doesn't increase as much as using WiFi due to the nature of master-slave communication in Bluetooth. Bluetooth is a Time Division Multiplexed (TDM) system, with a basic time unit of operation of $625\ \mu s$. A master node only transmits packets from even time slots, and slave nodes only respond on odd time slots. This strict TDM scheme offsets the increment in the average

throughput, because nodes can only send packets at the time slots they are assigned to. They cannot send when new packets are available if it is not their slots, and they have to wait till their slots.

We then conducted a TCP simulation. The simulation results, as shown in Fig. 6.1, suggest that transport availability queries do not significantly impact FTP flow. By increasing the transport availability query interval from 0.5 seconds to 20 seconds, the average throughput only increased by 2.28% over WiFi and by 1.59% over Bluetooth. Longer query intervals don't bring much benefit to data transfer in this situation since the time spent at the TCP layer becomes the dominating factor on transmission delay.

6.2 Query Overhead From Multiple Communicating Nodes

In this section, we examine the overhead of transport availability queries on data transfer when there are multiple pairs of communicating nodes and all of them conduct periodic queries. This is to evaluate the impact of availability queries on data transfer in the scenario of mesh networking. Although we are still dealing with single-hop communications in this section, the overhead incurred from periodic queries behaves the same as in mesh networking.

In our simulation, we increase the number of communicating heterogeneous node pairs from 1 to 5. Node 1 and node 2 communicate with each other through WiFi. Nodes in other pairs communicate via Bluetooth. We turn on the AFH function of Bluetooth in this scenario so that there is no interference between Bluetooth and WiFi. Using this setup, we can evaluate the impact of transport queries on WiFi throughput without sending all data on WiFi, which will drown out the effects of the overhead. All pairs of heterogeneous nodes conduct periodic transport availability queries at an interval of 1 second. As shown in Fig. 6.2, the distance of the two nodes within one pair is 5 meters, and the distance between neighboring pairs is 1 meter. All nodes are within the range of each other's transports. The simulations last for 20 seconds, and the data shown in Fig. 6.3 and Fig. 6.4 are collected at node 2.

When using CBR as the traffic generator and UDP as the transport layer protocol,

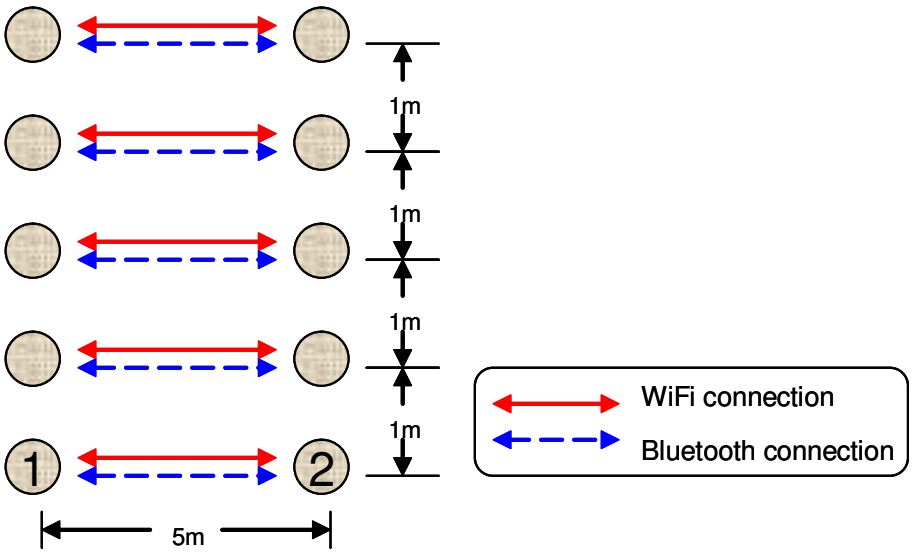

Figure 6.2: Query Overhead Between Two Communicating Nodes

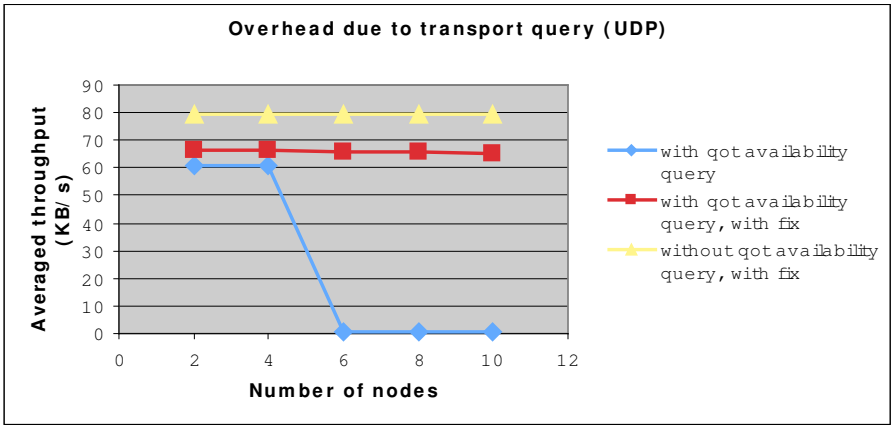

Figure 6.3: Query Overhead between Multiple Communicating Nodes

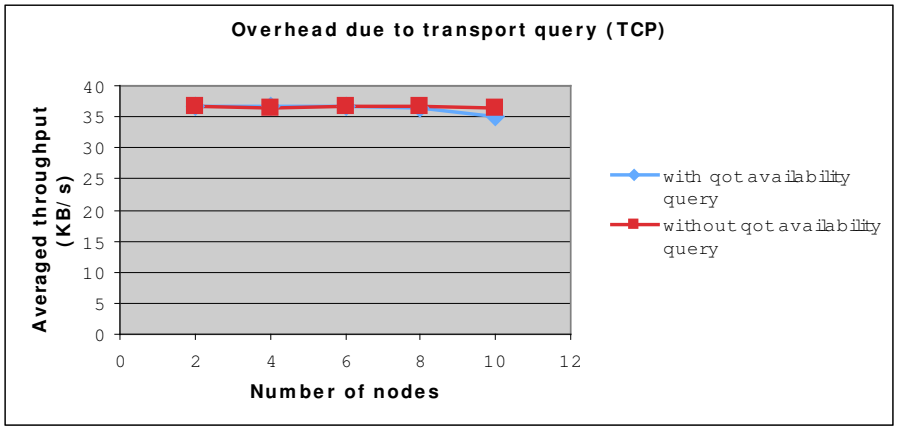

Figure 6.4: Query Overhead between Multiple Communicating Nodes

as shown in Fig. 6.3, there is a significant difference in the average throughput between communications with and without transport availability querying. With 5 pairs of nodes conducting periodic transport availability queries on WiFi, the average throughput drops by 17.93%. This is because QoT stops data transfer while conducting transport availability queries.

As shown in Fig. 6.3, throughput drops to nearly zero when WiFi becomes crowded. This drop occurs because UDP is an unreliable transport protocol and packets may be lost due to WiFi collisions. If QoT's synchronization packet is lost, a dead_lock may result between node 1 and node 2. At node 1, QoT won't remove packets from the buffer until it receives a synchronization response message, but if the synchronization request packet is lost, QoT never removes anything from the buffer, and the node cannot send packets out any more. A solution to this problem is for QoT to not buffer data packets from the session layer if the underlying transport is unreliable. When using UDP, applications should be responsible for providing reliability. Thus there is no need for packet synchronization, and this problem can be avoided.

We then conducted a TCP simulation. Simulation results, as shown in Fig. 6.4, suggest that there is no significant difference in the average throughput whether nodes conduct periodic transport availability queries or not when there are fewer than 4 pairs of

nodes. This is because the number of nodes that conduct periodic queries is small and the WiFi transport is not crowded, so the time spent at the TCP layer is the dominating factor for packet transmission delay. When more nodes perform transport availability querying, WiFi becomes more crowded and throughput drops. This is why we can observe a further drop of 3.93% in the average throughput when we employ 5 pairs of nodes. We can expect that this value becomes even lower when the number of WiFi node pairs increases.

Chapter 7

Conclusions and Future Work

In this paper, we introduced the value of constructing ad hoc networks by employing devices with multiple transports. Utilizing dynamic transport switching can provide better connectivity in ad hoc networks.

We used QoT as an example dynamic transport switching mechanism, and demonstrate that it can effectively mitigate the negative consequences of congestion and interference that may occur in ad hoc networks.

Despite its benefits, a dynamic transport switching mechanism may also incur overhead that limits data transfer. Since the example transport switching mechanism we use in this paper would freeze data transfer while performing transport availability querying, data throughput may drop due to such queries.

We also addressed two problems found in QoT and presented preliminary solutions to improve performance. In order to solve the problem that occurs when QoT conducts upgrade transport switching, we proposed to let QoT request a packet synchronization immediately after it switches data communication to a new transport. To solve the problem in the QoT output buffer, we proposed to not buffer session layer data packets at the QoT layer when the underlying transport protocol is unreliable. But QoT still needs to buffer session layer data packets when dealing with reliable transport layer protocols, such as TCP, or it won't know where to continue data communication upon a transport switch. Simulation results suggest that the proposed solution can effectively improve performance.

In this paper, we demonstrated the efficacy of dynamic transport switching for

single-hop connections. For future research, we are pursuing work on issues that may arise from multi-hop connections, such as how to conduct heterogeneous routing and how to intelligently balance work loads between different sources.

Appendix A

Heterogeneous Nodes and Dynamic Transport Switching in NS-2

This document describes the files comprising the heterogeneous nodes and dynamic transport switching mechanism in NS-2. This document also provides guidelines on how to configure simulations involving heterogeneous nodes in NS-2.

A.1 File Layout

The files comprising the heterogeneous nodes and dynamic transport switching mechanism are organized under the `/ns-2.28/qot` directory.

The `qot` directory includes the following files and subdirectory.

- `qot.h`
- `hdr_qot.h`
- `ns-qot.tcl`
- `qot.cc`
- `qot-node.cc`
- `qot_queue.cc`
- `qot_timers.cc`
- `dm`

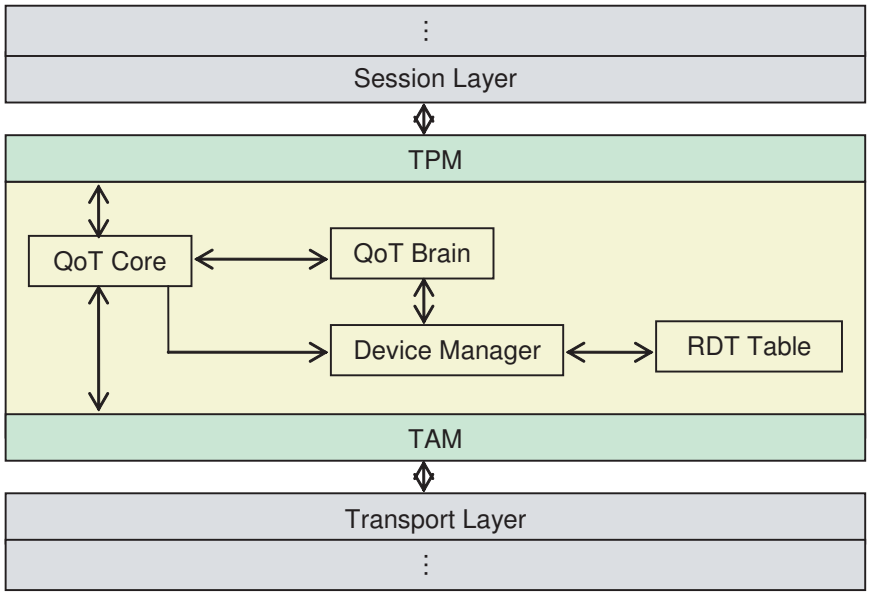

Figure A.1: QoT Architecture

File `qot-node.cc` contains functions that constitute the `QoT Core` module in QoT architecture, as demonstrated in Fig.A.1. File `qot.cc` contains functions that constitute the `QoT Brain`, `Device Manager`, `TPM`, `TAM`, and `RDT Table` modules. File `qot_queue.cc` contains functions that implement QoT data buffer. File `qot_timers.cc` includes functions that deploy QoT internal timers. File `ns-qot.tcl` includes QoT interface functions to `Tcl` space. Directory `dm` includes files that deploy decision making mechanisms in `QoT Brain`.

Interfacing functions to the original NS-2 architecture are also implemented, and the files under the following directories are significantly modified.

- `tcl/lib`: `ns-lib.tcl / ns-mobilenode.tcl / ns-packet.tcl`
- `apps`: `app.cc / app.h / udp.cc`
- `aodv`: `aodv.cc`
- `bluetooth`: `baseband.cc / baseband.h / ns-btnode.tcl`

- common: agent.cc / agent.h / mobilenode.h / packet.h
- mac: mac-802_11.cc / mac-802_11.h / mac.h / mac-wu.h
- queue: queue.h
- tcp: tcp-full.cc / tcp-sink.cc
- wpan: p802_15_4mac.cc
- wu: wu.cc

A.2 Simulations with Heterogeneous Nodes in NS-2

Tcl commands are implemented to let users be able to manage simulations that involve heterogeneous nodes in NS-2. This section describes the usages of these Tcl commands.

Since the heterogeneous nodes are modeled after QoT, creating a heterogeneous node in a simulation is equivalent to creating a QoT node in NS-2. To create a QoT node, the user needs to turn the qot flag on in the nodeconfig function of Tcl class Simulator. Following is an example of creating a QoT node in a simulation.

```
$ns_ node-config -qot ON
set qot_node(0) [$ns_ node]
set qot_node(1) [$ns_ node]
$ns_ node-config -qot OFF
```

In the example above, $ns_ is an instance of the Tcl class Simulator. After the qot flag is turned on, creating a QoT node is the same as creating an ordinary homogeneous node in the original NS-2 distribution. In this example, two QoT nodes are created. The qot flag is turned off after the creation of the QoT nodes in order to create potential subsequent homogeneous nodes in the simulation.

After a QoT node is created, transports need to be attached to it. Recall that heterogeneous nodes are "virtual" in the sense that they don't possess traditional underlying protocol stack layers. Homogeneous nodes are created as the transports of a heterogeneous node. Transports are attached to a QoT node by the command `attach_transport`.

```
$qot_node(0) attach_transport BT $bt_node(0) TCP $tcp0
AODV
$qot_node(0) attach_transport WIFI $wifi_node(0) TCP
$tcp2 AODV
```

The example above attaches two transports, Bluetooth and WiFi, to a QoT node. `BT` and `WIFI` inform the QoT node the type of the transports attached. `$bt_node(0)` and `$wifi_node(0)` are handlers to the two transports, which are actually two homogeneous nodes in NS-2. `$tcp0` and `$tcp2` are handlers of the transport layers of the transports Bluetooth and WiFi respectively. AODV identifies the type of the routing agent that the transports support.

Once QoT nodes are created, the user can use command `connect-qot-node` in `Tcl` class `Simulator` to connect them.

```
$ns_ connect-qot-node $qot_node(0) $qot_node(1)
```

In order to realize the dynamic switching of data traffic between transports, traffic generators are associated with the TPM module of QoT.

```
set ftp [new Application/FTP]
set tpm1 [new Agent/QTPM]
$ftp attach-qot $qot_node(0) $tpm1
```

The example above first creates a FTP instance and a TPM module instance. The traffic generator is then linked to the TPM module texttt$tpm1 of QoT node `$qot_node(0)`. Command `attach-qot` is implemented in the `Application` class, so that commonly used traffic generators, such as FTP and CBR, automatically inherit this command.

To maintain an appearance of a single node in the simulation, a QoT node synchronizes the coordinates of all its transports by the command `setLocation`.

```
$qot_node(0) setLocation 5.0 5.0 0.0
```

In the above example, `$qot_node(0)` set the coordinates of all its transports to 5.0 5.0 0.0.

QoT nodes can also move in a simulation. All the transports of a QoT node move to a new destination from a common starting point at a same speed.

```
$ns_ at 15.0 "$qot_node(0) setdest 10.0 6.0 1.0"
```

In above example, `$qot_node(0)` starts moving to (10.0, 6.0) with a speed of 1.0 *m/s*.

A QoT node can conduct transport switching by command `transport_switch`.

```
$ns_ at 14.0 "$qot_node(0) transport_switch
[$qot_node(1) set id_] BT"
```

In the example above, `$qot_node(0)` communicates with `$qot_node(1)`, and switches the active transport to Bluetooth at second 14.0.

Two statistical functions implemented in class `QoTNode` are also provided to help users collect simulation results. Command `printThroughput` prints the throughput over time collected at the QoT layer on the receiving node. Command `printPower` prints the power consumption over time of the QoT node.

Appendix B

Heterogeneous Node Reference Manual

B.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

B.2 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

B.3 act_trans_list Struct Reference

Active transport link list.

`#include <qot.h>`

Collaboration diagram for act_trans_list:

Public Attributes

- sharedT * share

  Pointer to the shared transport.

- act_trans_list * next_

  Linkage to the next entry on the link list.

- act_trans_list * prev_

  Linkage to the previous entry on the link list.

B.3.1 Detailed Description

Active transport link list.

This link list is maintained and used by QoTBrain for decision making purpose. This list might be the sharedT list of the DevTabEntry or its subset. This list should always be sorted and the most desired active transport should be placed to the head.

B.3.2 Member Data Documentation

B.3.2.1 act_trans_list* act_trans_list::next_

Linkage to the next entry on the link list.

B.3.2.2 act_trans_list* act_trans_list::prev_

Linkage to the previous entry on the link list.

B.3.2.3 sharedT* act_trans_list::share

Pointer to the shared transport.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.4 app_data Struct Reference

The application data that a session layer protocol sends down to the QoT layer.

`#include <qot.h>`

Collaboration diagram for app_data:

Public Attributes

- int size

  The size of this application data packet, in bytes.

- AppData * dat
- const char * flag
- app_data * next_

  Linkage to the next application data packet in the application packets link list of the QoT node.

- nsaddr_t dest

  The address of session packets destination.

- double ts_

B.4.1 Detailed Description

The application data that a session layer protocol sends down to the QoT layer.

B.4.2 Member Data Documentation

B.4.2.1 AppData* app_data::dat

B.4.2.2 nsaddr_t app_data::dest

The address of session packets destination.

B.4.2.3 const char* app_data::flag

B.4.2.4 app_data* app_data::next_

Linkage to the next application data packet in the application packets link list of the QoT node.

Session packets are buffered at the QoT layer before sending out. Upon receiving session layer packets, the Qot layer first check if there already exists a QoT connection for this destination. If so, QoT segments the session packets into QoT packets and put them into the QoT output buffer waiting to be sent out (if with TCP). If not, QoT put the session packets in a buffer and start establishing the required connection.

B.4.2.5 int app_data::size

The size of this application data packet, in bytes.

B.4.2.6 double app_data::ts_

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.5 CallBack Struct Reference

Callback link list entry.

`#include <qot.h>`

Collaboration diagram for CallBack:

Public Attributes

- char * call_type

  Callback type: DATA_SYNC, PERIODIC_QUERY, SWITCH_REQUEST, DATA_SEND.

- sharedT * share

  Pointer to a shared transport.

- nsaddr_t remote_id

  The address of the remote device.

- DevTabEntry * entry

  Pointer to an entry of the Remote Device Table.

- int pkt_id

  QoT data packet sequence number.

- CallBack * next_

  Pointer to the next item in the callback link list.

B.5.1 Detailed Description

Callback link list entry.

B.5.2 Member Data Documentation

B.5.2.1 char* CallBack::call_type

Callback type: DATA_SYNC, PERIODIC_QUERY, SWITCH_REQUEST, DATA_SEND.

B.5.2.2 DevTabEntry* CallBack::entry

Pointer to an entry of the Remote Device Table.

B.5.2.3 CallBack* CallBack::next_

Pointer to the next item in the callback link list.

B.5.2.4 int CallBack::pkt_id

QoT data packet sequence number.

B.5.2.5 nsaddr_t CallBack::remote_id

The address of the remote device.

B.5.2.6 sharedT* CallBack::share

Pointer to a shared transport.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.6 DevTabEntry Class Reference

Entry in the Remote Device Table.

`#include <qot.h>`

Collaboration diagram for DevTabEntry:

Public Member Functions

- DevTabEntry (RDT ∗rdt)

  Class constructor.

- nsaddr_t remote_id ()

  Retrieve the ID of the communicating node.

- void set_id (nsaddr_t id)

  Set the ID of the communicating node.

- void set_roll (int r)

  Set the roll of this node in the communication.

- int roll ()

  Retrieve the roll of this node.

- sharedT ∗ currentT ()

  Retrieve the pointer to the transport that is being used for communication.

- RDT ∗ rdt ()

Pointer to the Remote Device Table.

- void setRDT (RDT ∗rdt)

  Set the pointer to the Remote Device Table.

- void setCurrent (sharedT ∗t)

  Set the pointer to the active transport.

- void setOldCurrent (sharedT ∗t)

  Set the pointer to the transport that was being used for the communication before the transport switching.

- sharedT ∗ oldCurrent ()

  Retrieve the pointer to the transport that was being used for the communication before the transport switching.

- int received ()
- void setReceived (int i)
- int queLen ()

  Retrieve the length of the queue.

- QoTState qotState ()

  Retrieve the current state of QoT.

- void setState (QoTState st)

  Set the state of QoT.

- app_data ∗ check_app_buff ()

  Retrieve the application packet buffer.

- act_trans_list ∗ get_active_by_tag (char ∗tag)

  Retrieve the active transport with the specified type.

- sharedT ∗ getAvailShare ()

  Retriee the first available transport in the shared transport link list.

- unsigned getPktID ()

  Get the next QoT data packet ID.

- void handleAppData (app_data ∗d)

  Process the received session layer data packets.

- void createQoTHdr (int size, app_data ∗d)

  Create QoT data packet header.

- sharedT ∗ getT (char ∗tag, ns_addr_t dest)
- sharedT ∗ hasT (char ∗tag)

  Check if there exists a shared transport with the specified type in the link list.

Public Attributes

- RDT ∗ rdt_

  Pointer to the Remote Device Table that this table entry belongs to.

Private Member Functions

- int removeT (transport_stack ∗r)

Remove the shared transport from the link list.

- void syncCheck (unsigned id)

  Request a data synchronization to the send buffer.

- void insertActiveTransport (sharedT ∗share)

  Insert the transport into the active transport list.

- void removeActiveTransport (sharedT ∗share)

  Remove the transport into the active transport list.

- void updateQueryResults (sharedT ∗share, int result)
- void sortActiveTransport ()

  Sort the active transports from high to low according to their utilities.

Private Attributes

- nsaddr_t devID

  The address of the remote node.

- int query_complete

  A flag that identifies if the transport availability query for this remote device is done.

- QoTOutQueue ∗ que

  Pointer to the send buffer for this connection.

- sharedT ∗ currentT_

  Pointer to the active transport.

- sharedT ∗ old_currentT

  Pointer to the transport that was used before the transport switching attempt.

- sharedT ∗ share_t_hdr

  Head of the link list of the shared transports of a QoT connection.

- sharedT ∗ share_t_tail

  Tail of the link list of the shared transport of a QoT connection.

- int roll_

  The roll of the node in a QoT connection.

- act_trans_list ∗ list_head

  Head of the available shared transport link list.

- int rcvd_
- QoTState state

  The QoT state for this connection.

- int infinite_send_

  A flag that identifies if the traffic generator issues infinite packet send.

- app_data ∗ app_buff

  A buffer to store session data packet information.

- unsigned qot_pkt_id

  Sequence number of QoT data packet.

- DevTabEntry * next_

  Pointer to the next device table entry on the link list.

- DevTabEntry * prev_

  Pointer to the previous device table entry on the link list.

Friends

- class RDT
- class QoTBrain
- class QoTNode
- class QoTOutQueue
- class TransportQueryTimer
- class SyncTimer

B.6.1 Detailed Description

Entry in the Remote Device Table.

QoT creates an entry in the Remote Device Table for each communicating node.

B.6.2 Constructor & Destructor Documentation

B.6.2.1 DevTabEntry::DevTabEntry (RDT * *rdt*)

Class constructor.

B.6.3 Member Function Documentation

B.6.3.1 app_data* DevTabEntry::check_app_buff () [inline]

Retrieve the application packet buffer.

B.6.3.2 void DevTabEntry::createQoTHdr (int *size*, app_data * *d*)

Create QoT data packet header.

This function creates the QoT headers for QoT data packets. The actual packets are created at the transport layer. The headers created here are copied to the corresponding fields of the actual packets after they are created.

Parameters:

size The size of the QoT data packet that is to be created.

d The session layer data packet.

B.6.3.3 sharedT* DevTabEntry::currentT () [inline]

Retrieve the pointer to the transport that is being used for communication.

B.6.3.4 act_trans_list* DevTabEntry::get_active_by_tag (char * *tag*)

Retrieve the active transport with the specified type.

If no transport with the specified type found in the list, NULL is returned.

Parameters:

tag The type of the transport, such as WIFI or BT.

B.6.3.5 sharedT∗ DevTabEntry::getAvailShare ()

Retriee the first available transport in the shared transport link list.

B.6.3.6 unsigned DevTabEntry::getPktID () [inline]

Get the next QoT data packet ID.

B.6.3.7 sharedT∗ DevTabEntry::getT (char ∗ *tag*, ns_addr_t *dest*)

Return the pointer to the shared pointer that meets the specified type and address. Create a new shared transport entry with the specified type and address if no transport is found. Correponding fields in the entry are filled. The newly created entry is appended to the link list.

Parameters:

tag The type of the shared transport, such as WIFI or BT.

dest The address and the port number of the corresponding transport on the remote node.

Returns:

The pointer to the transport with the specified type and address.

B.6.3.8 void DevTabEntry::handleAppData (app_data * *d*)

Process the received session layer data packets.

Segment the sessoin layer data packet into QoT data packets. If the infinite_send_ flag is set, QoT data packets will be generated till the queue is full.

Parameters:

d Session layer data packet.

B.6.3.9 sharedT* DevTabEntry::hasT (char * *tag*)

Check if there exists a shared transport with the specified type in the link list.

Parameters:

The type of the shared transport, such as WIFI or BT.

Returns:

The pointer to the transport with the specified type or NULL.

B.6.3.10 void DevTabEntry::insertActiveTransport (sharedT * *share*) [private]

Insert the transport into the active transport list.

Parameters:

The shared transport that is to be inserted.

B.6.3.11 sharedT* DevTabEntry::oldCurrent () [inline]

Retrieve the pointer to the transport that was being used for the communication before the transport switching.

B.6.3.12 QoTState DevTabEntry::qotState () [inline]

Retrieve the current state of QoT.

B.6.3.13 int DevTabEntry::queLen () [inline]

Retrieve the length of the queue.

B.6.3.14 RDT* DevTabEntry::rdt () [inline]

Pointer to the Remote Device Table.

B.6.3.15 int DevTabEntry::received () [inline]

B.6.3.16 nsaddr_t DevTabEntry::remote_id () [inline]

Retrieve the ID of the communicating node.

B.6.3.17 void DevTabEntry::removeActiveTransport (sharedT * *share*) [private]

Remove the transport into the active transport list.

Parameters:

The shared transport that is to be removed.

B.6.3.18 int DevTabEntry::removeT (transport_stack * *r*) [private]

Remove the shared transport from the link list.

Parameters:

r The protocol stack of the transport that is to be removed from the link list.

Returns:

0 if there is no more shared transport after the removal. 1 if removal operation failed.

B.6.3.19 int DevTabEntry::roll () [inline]

Retrieve the roll of this node.

B.6.3.20 void DevTabEntry::set_id (nsaddr_t *id*) [inline]

Set the ID of the communicating node.

B.6.3.21 void DevTabEntry::set_roll (int *r*) [inline]

Set the roll of this node in the communication.

1 for master node, 0 for slave node.

B.6.3.22 void DevTabEntry::setCurrent (sharedT * *t*) [inline]

Set the pointer to the active transport.

B.6.3.23 void DevTabEntry::setOldCurrent (sharedT * *t*) [inline]

Set the pointer to the transport that was being used for the communication before the transport switching.

This pointer is used for fallback in case of a failed upgrade transport switching.

B.6.3.24 void DevTabEntry::setRDT (RDT * *rdt*) [inline]

Set the pointer to the Remote Device Table.

B.6.3.25 void DevTabEntry::setReceived (int *i*) [inline]

B.6.3.26 void DevTabEntry::setState (QoTState *st*) [inline]

Set the state of QoT.

B.6.3.27 void DevTabEntry::sortActiveTransport () [private]

Sort the active transports from high to low according to their utilities.

B.6.3.28 void DevTabEntry::syncCheck (unsigned *id*) [inline, private]

Request a data synchronization to the send buffer.

Parameters:

id The sequence number of the data packet that is last received by the remote node.

B.6.3.29 void DevTabEntry::updateQueryResults (sharedT * *share*, int *result*) [private]

B.6.4 Friends And Related Function Documentation

B.6.4.1 friend class QoTBrain [friend]

B.6.4.2 friend class QoTNode [friend]

B.6.4.3 friend class QoTOutQueue [friend]

B.6.4.4 friend class RDT [friend]

B.6.4.5 friend class SyncTimer [friend]

B.6.4.6 friend class TransportQueryTimer [friend]

B.6.5 Member Data Documentation

B.6.5.1 app_data* DevTabEntry::app_buff [private]

A buffer to store session data packet information.

B.6.5.2 sharedT* DevTabEntry::currentT_ [private]

Pointer to the active transport.

B.6.5.3 nsaddr_t DevTabEntry::devID [private]

The address of the remote node.

B.6.5.4 int DevTabEntry::infinite_send_ [private]

A flag that identifies if the traffic generator issues infinite packet send.

B.6.5.5 act_trans_list* DevTabEntry::list_head [private]

Head of the available shared transport link list.

B.6.5.6 DevTabEntry* DevTabEntry::next_ [private]

Pointer to the next device table entry on the link list.

B.6.5.7 sharedT* DevTabEntry::old_currentT [private]

Pointer to the transport that was used before the transport switching attempt.

B.6.5.8 DevTabEntry* DevTabEntry::prev_ [private]

Pointer to the previous device table entry on the link list.

B.6.5.9 unsigned DevTabEntry::qot_pkt_id [private]

Sequence number of QoT data packet.

Start from 0.

B.6.5.10 QoTOutQueue* DevTabEntry::que [private]

Pointer to the send buffer for this connection.

B.6.5.11 int DevTabEntry::query_complete [private]

A flag that identifies if the transport availability query for this remote device is done.

B.6.5.12 int DevTabEntry::rcvd_ [private]

B.6.5.13 RDT* DevTabEntry::rdt_

Pointer to the Remote Device Table that this table entry belongs to.

B.6.5.14 int DevTabEntry::roll_ [private]

The roll of the node in a QoT connection.

1 for master, 0 for slave.

B.6.5.15 sharedT* DevTabEntry::share_t_hdr [private]

Head of the link list of the shared transports of a QoT connection.

B.6.5.16 sharedT* DevTabEntry::share_t_tail [private]

Tail of the link list of the shared transport of a QoT connection.

B.6.5.17 QoTState DevTabEntry::state [private]

The QoT state for this connection.

The documentation for this class was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.7 p_consumption Struct Reference

The power consumption link list item used in the statistical timer.

`#include <qot.h>`

Collaboration diagram for p_consumption:

Public Attributes

- p_consumption * next

  Pointer to the next item in the link list.

- double p_value

  Power consumption value.

- double t

  The time period that the collected data correspond to.

B.7.1 Detailed Description

The power consumption link list item used in the statistical timer.

B.7.2 Member Data Documentation

B.7.2.1 p_consumption* p_consumption::next

Pointer to the next item in the link list.

B.7.2.2 double p_consumption::p_value

Power consumption value.

B.7.2.3 double p_consumption::t

The time period that the collected data correspond to.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.8 prio_info Struct Reference

Transport priority information.

`#include <hdr.h>`

Collaboration diagram for prio_info:

Public Attributes

- char * tag

  Transport type.

- double utility

  Transport utility.

- prio_info * next
- prio_info * prev

B.8.1 Detailed Description

Transport priority information.

B.8.2 Member Data Documentation

B.8.2.1 prio_info* prio_info::next

B.8.2.2 prio_info* prio_info::prev

B.8.2.3 char* prio_info::tag

Transport type.

B.8.2.4 double prio_info::utility

Transport utility.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.9 qot_con_acc Struct Reference

QOT_CONNECT_ACCEPT message.

`#include <hdr.h>`

B.9.1 Detailed Description

QOT_CONNECT_ACCEPT message.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.10 qot_con_rej Struct Reference

QOT_CONNECT_REJECT message.

`#include <hdr.h>`

Public Attributes

- QoTReason reason

  Reject reason.

B.10.1 Detailed Description

QOT_CONNECT_REJECT message.

B.10.2 Member Data Documentation

B.10.2.1 QoTReason qot_con_rej::reason

Reject reason.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.11 qot_con_req Struct Reference

QOT_CONNECT_REQUEST message.

`#include <hdr.h>`

Public Attributes

- char * tag

  Transport type.

B.11.1 Detailed Description

QOT_CONNECT_REQUEST message.

B.11.2 Member Data Documentation

B.11.2.1 char* qot_con_req::tag

Transport type.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.12 qot_data_snd Struct Reference

QOT_DATA_SEND message.

`#include <hdr.h>`

Public Attributes

- const char * app_flag
- AppData * app_data
- unsigned qot_pkt_id

  Qot data packet id.

B.12.1 Detailed Description

QOT_DATA_SEND message.

B.12.2 Member Data Documentation

B.12.2.1 AppData* qot_data_snd::app_data

B.12.2.2 const char* qot_data_snd::app_flag

B.12.2.3 unsigned qot_data_snd::qot_pkt_id

Qot data packet id.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.13 qot_data_sync_pnt Struct Reference

QOT_DATA_SYNC_POINT message.

`#include <hdr.h>`

Public Attributes

- int credit
- unsigned pkt_id

  The ID of the packet that is last received.

B.13.1 Detailed Description

QOT_DATA_SYNC_POINT message.

B.13.2 Member Data Documentation

B.13.2.1 int qot_data_sync_pnt::credit

B.13.2.2 unsigned qot_data_sync_pnt::pkt_id

The ID of the packet that is last received.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.14 qot_data_sync_req Struct Reference

QOT_DATA_SYNC_REQUEST message.

`#include <hdr.h>`

B.14.1 Detailed Description

QOT_DATA_SYNC_REQUEST message.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.15 qot_discon_acc Struct Reference

QOT_DISCONNECT_ACCEPT message.

`#include <hdr.h>`

B.15.1 Detailed Description

QOT_DISCONNECT_ACCEPT message.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.16 qot_discon_req Struct Reference

QOT_DISCONNECT_REQUEST message.

`#include <hdr.h>`

B.16.1 Detailed Description

QOT_DISCONNECT_REQUEST message.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.17 qot_rem_acc Struct Reference

QOT_RESUME_ACCEPT messsage.

`#include <hdr.h>`

Public Attributes

- int sync_point

B.17.1 Detailed Description

QOT_RESUME_ACCEPT messsage.

B.17.2 Member Data Documentation

B.17.2.1 int qot_rem_acc::sync_point

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.18 qot_rem_rej Struct Reference

QOT_RESUME_REJECT message.

`#include <hdr.h>`

B.18.1 Detailed Description

QOT_RESUME_REJECT message.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.19 qot_rem_req Struct Reference

QOT_RESUME_REQUEST message.

`#include <hdr.h>`

Public Attributes

- int sync_point

  The sequence number of the qot data packet from where the connection should be resumed.

B.19.1 Detailed Description

QOT_RESUME_REQUEST message.

B.19.2 Member Data Documentation

B.19.2.1 int qot_rem_req::sync_point

The sequence number of the qot data packet from where the connection should be resumed.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.20 qot_stack Union Reference

The union of possible transport stacks of a QoT node.

`#include <qot.h>`

Collaboration diagram for qot_stack:

Public Attributes

- stack_bt bt
- stack_wifi wifi
- stack_zigbee zigbee
- stack_wusb wusb

B.20.1 Detailed Description

The union of possible transport stacks of a QoT node.

B.20.2 Member Data Documentation

B.20.2.1 stack_bt qot_stack::bt

B.20.2.2 stack_wifi qot_stack::wifi

B.20.2.3 stack_wusb qot_stack::wusb

B.20.2.4 stack_zigbee qot_stack::zigbee

The documentation for this union was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.21 qot_swh_acc Struct Reference

QOT_SWITCH_ACCEPT message.

`#include <hdr.h>`

Public Attributes

- char * tag

  Transport type.

B.21.1 Detailed Description

QOT_SWITCH_ACCEPT message.

B.21.2 Member Data Documentation

B.21.2.1 char* qot_swh_acc::tag

Transport type.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.22 qot_swh_qry Struct Reference

QOT_SWITCH_QUERY message.

`#include <hdr.h>`

Public Attributes

- SwitchType type

  Switch type, upgrade switching or downgrade switching.

B.22.1 Detailed Description

QOT_SWITCH_QUERY message.

B.22.2 Member Data Documentation

B.22.2.1 SwitchType qot_swh_qry::type

Switch type, upgrade switching or downgrade switching.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.23 qot_swh_qry_rep Struct Reference

QOT_SWITCH_QUERY_RESPONSE.

`#include <hdr.h>`

Collaboration diagram for qot_swh_qry_rep:

Public Attributes

- prio_info * pri_list

  Transport information link list.

B.23.1 Detailed Description

QOT_SWITCH_QUERY_RESPONSE.

B.23.2 Member Data Documentation

B.23.2.1 prio_info* qot_swh_qry_rep::pri_list

Transport information link list.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.24 qot_swh_rej Struct Reference

QOT_SWITCH_REJECT message.

`#include <hdr.h>`

Public Attributes

- char * tag

  Transport type.

B.24.1 Detailed Description

QOT_SWITCH_REJECT message.

B.24.2 Member Data Documentation

B.24.2.1 char* qot_swh_rej::tag

Transport type.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.25 qot_swh_req Struct Reference

QOT_SWITCH_REQUEST.

`#include <hdr.h>`

Public Attributes

- char * tag

  Transport type.

B.25.1 Detailed Description

QOT_SWITCH_REQUEST.

B.25.2 Member Data Documentation

B.25.2.1 char* qot_swh_req::tag

Transport type.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.26 qot_trans_info_qry Struct Reference

QOT_TRANSPORT_INFO_QUERY messge content.

`#include <hdr.h>`

Public Attributes

- char * tag
- int count
- double SNR_request
- double signal_request

B.26.1 Detailed Description

QOT_TRANSPORT_INFO_QUERY messge content.

B.26.2 Member Data Documentation

B.26.2.1 int qot_trans_info_qry::count

B.26.2.2 double qot_trans_info_qry::signal_request

B.26.2.3 double qot_trans_info_qry::SNR_request

B.26.2.4 char* qot_trans_info_qry::tag

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.27 qot_trans_info_qry_rep Struct Reference

QOT_TRANSPORT_INFO_QUERY_RESPONSE messge content.

`#include <hdr.h>`

Public Attributes

- int count
- double SNR_request
- double SNR_response
- double signal_request
- double signal_response

B.27.1 Detailed Description

QOT_TRANSPORT_INFO_QUERY_RESPONSE messge content.

B.27.2 Member Data Documentation

B.27.2.1 int qot_trans_info_qry_rep::count

B.27.2.2 double qot_trans_info_qry_rep::signal_request

B.27.2.3 double qot_trans_info_qry_rep::signal_response

B.27.2.4 double qot_trans_info_qry_rep::SNR_request

B.27.2.5 double qot_trans_info_qry_rep::SNR_response

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.28 qot_trans_qry Struct Reference

QOT_TRANSPORT_QUERY message.

`#include <hdr.h>`

B.28.1 Detailed Description

QOT_TRANSPORT_QUERY message.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.29 qot_trans_qry_rep Struct Reference

QOT_TRANSPORT_QUERY_RESPONSE messge content.

`#include <hdr.h>`

Collaboration diagram for qot_trans_qry_rep:

Public Attributes

- int num
- trans_info * head

B.29.1 Detailed Description

QOT_TRANSPORT_QUERY_RESPONSE messge content.

B.29.2 Member Data Documentation

B.29.2.1 trans_info* qot_trans_qry_rep::head

B.29.2.2 int qot_trans_qry_rep::num

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.30 QoTBrain Class Reference

QoT Brain.

`#include <qot.h>`

Collaboration diagram for QoTBrain:

Public Member Functions

- QoTBrain ()

  Class constructor.

- void setnode (QoTNode *node)

  Set the pointer to the QoT node that this QoT brain belongs to.

- void settable (RDT *table)

  Set the pointer to the Remote Device Table that this QoT brain associates with.

- void startQuery (DevTabEntry *entry)

  Start transport availability query.

- void activeHeadChanged (DevTabEntry *entry, SwitchType type)

Static Public Member Functions

- static void calculateUtilitybyPower (sharedT *)

  Calculate the utility of a transport by its power consumption.

- static void calculateUtilitybyDatarate (sharedT *)

Calculate the utility of a transport by its data rate.

Public Attributes

- void(∗ getUtility)(sharedT ∗)

Private Attributes

- QoTNode ∗ node_

  Pointer to the QoT node that this QoT brain belongs to.

- RDT ∗ rd_table

  Pointer to the Remote Device Table that this QoT brain associates with.

B.30.1 Detailed Description

QoT Brain.

B.30.2 Constructor & Destructor Documentation

B.30.2.1 QoTBrain::QoTBrain ()

Class constructor.

B.30.3 Member Function Documentation

B.30.3.1 void QoTBrain::activeHeadChanged (DevTabEntry * *entry*, SwitchType *type*)

B.30.3.2 static void QoTBrain::calculateUtilitybyDatarate (sharedT *) [static]

Calculate the utility of a transport by its data rate.

B.30.3.3 static void QoTBrain::calculateUtilitybyPower (sharedT *) [static]

Calculate the utility of a transport by its power consumption.

B.30.3.4 void QoTBrain::setnode (QoTNode * *node*) [inline]

Set the pointer to the QoT node that this QoT brain belongs to.

B.30.3.5 void QoTBrain::settable (RDT * *table*) [inline]

Set the pointer to the Remote Device Table that this QoT brain associates with.

B.30.3.6 void QoTBrain::startQuery (DevTabEntry * *entry*)

Start transport availability query.

Parameters:

entry An RDT entry that includes shared transports that need to conduct availability queries.

B.30.4 Member Data Documentation

B.30.4.1 void(∗ QoTBrain::getUtility)(sharedT ∗)

B.30.4.2 QoTNode∗ QoTBrain::node_ [private]

Pointer to the QoT node that this QoT brain belongs to.

B.30.4.3 RDT∗ QoTBrain::rd_table [private]

Pointer to the Remote Device Table that this QoT brain associates with.

The documentation for this class was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.31 QoTNode Class Reference

QoT node.

`#include <qot.h>`

Collaboration diagram for QoTNode:

Public Member Functions

- QoTNode ()

  Class constructor.

- int command (int argc, const char ∗const ∗argv)

  Command function that implements Tcl-C++ interfacing functions.

- transport_stack ∗ get_trans_by_tag (char ∗s)

  Retrieve a transport stack according to the provided string.

- hdr_qot ∗ getDummy ()

  Retrieve the QoT header stored in the dummy_.

- void clearDummy ()

  Clear the data stored in the dummy_.

- void setDummy (hdr_qot ∗hdr)

  Set the dummy_.

- void setTPM (QTPM ∗tpm)

Set the pointer to a TPM module.

- void setApp (Application ∗app)

  Set the pointer to the session layer protocol.

- nsaddr_t getDest ()

  Retrieve the address of the remote device.

- void checkRecvBuff (DevTabEntry ∗entry)
- void recv (Packet ∗p, transport_stack ∗stack)

  Receive incoming packets.

- void sendDown (transport_stack ∗s)

  Send a QoT packet down to the underlying protocol.

- void downRecv (int size, AppData ∗appdata, const char ∗flags=0)

  Receive a packet from the session layer protocol.

- void linkBreak (Packet ∗p)

  Link break handler.

- void printLocation ()

  Print the coordiates of the transports of this QoT node.

- void devDiscover (nsaddr_t dst)

  Discover remote device.

- void transQuery (nsaddr_t dst, transport_stack ∗stack)

QOT_TRANSPORT_QUERY.

- void transQueryResponse (Packet ∗p, transport_stack ∗stack)

  QOT_TRANSPORT_QUERY_RESPONSE.

- void transInfoQuery (sharedT ∗share, nsaddr_t dst)

  QOT_TRANSPORT_INFO_QUERY.

- void transInfoQueryResponse (Packet ∗p, transport_stack ∗stack, double request, double signal, int count)

  QOT_TRANSPORT_INFO_QUERY_RESPONSE.

- void connectRequest (sharedT ∗share, nsaddr_t dst)

  QOT_CONNECT_REQUEST.

- void connectAccept (sharedT ∗share, nsaddr_t dst)

  QOT_CONNECT_ACCEPT.

- void connectReject (Packet ∗p, transport_stack ∗stack, QoTReason r)

  QOT_CONNECT_REJECT.

- void dataSend (DevTabEntry ∗entry)

  QOT_DATA_SEND.

- void dataSync (DevTabEntry ∗entry)

  QOT_DATA_SYNC_POINT.

- void switchQuery (DevTabEntry ∗entry, SwitchType type)

QOT_SWITCH_QUERY.

- void switchQueryResponse (DevTabEntry ∗entry, nsaddr_t dst)

  QOT_SWITCH_QUERY_RESPONSE.

- void switchRequest (sharedT ∗share, nsaddr_t dst)

  QOT_SWITCH_REQUEST.

- void switchAccept (sharedT ∗share, nsaddr_t dst)

  QOT_SWITCH_ACCEPT.

- void switchReject (sharedT ∗share, nsaddr_t dst)

  QOT_SWITCH_REJECT.

- void resumeRequest (DevTabEntry ∗entry, nsaddr_t dst)

  QOT_RESUME_REQUEST.

- void resumeAccept (DevTabEntry ∗entry, nsaddr_t dst)

  QOT_RESUME_ACCEPT.

- void resumeReject (sharedT ∗share, nsaddr_t dst)

  QOT_RESUME_REJECT.

- void disconnectRequest (sharedT ∗share, nsaddr_t dst)

  QOT_DISCONNECT_REQUEST.

- void disconnectAccept (sharedT ∗share, nsaddr_t dst)

  QOT_DISCONNECT_ACCEPT.

- QoTOutQueue ∗ checkOutputBuff (nsaddr_t dest)

  Retrieve the send buffer for a connection.

- transport_stack ∗ getStack (char ∗tag)

  Retrieve a transport stack of the QoT node.

- void checkCallBack ()

  Check callback queue.

- void printQoTPacket (hdr_qot ∗hdr)

  Print QoT packet information.

Public Attributes

- int busy_

  The flag that identifies if the QoT node is performing transport availability query.

- StatTimer ∗ s_timer

  Statistical timer.

Private Attributes

- hdr_qot ∗ dummy_

  The buffer that transfers QoT packet header information to the transport agent when it creates a packet.

- nsaddr_t dst_

The address of the remote device.

- transport_stack ∗ trans_head

  Head of the transport stack link list.

- int transport_nn
- int do_query
- agent_value agent_v
- TransportQueryTimer ∗ tq_timer

  Transport query timer.

- QTPM ∗ tpm_

  Pointer to a TPM module.

- Application ∗ app_

  Pointer to the attached session layer protocol.

- QoTQueue ∗ recv_buff

  QoT receive buffer.

- RDT ∗ rd_table

  Pointer to the Remote Device Table.

- QoTBrain ∗ brain

  Pointer to the QoT brain.

- CallBack ∗ call_back_h

  Head of the callback link list.

- CallBack * call_back_t

  Tail of the callback link list.

Friends

- class QoTBrain
- class DevTabEntry
- class Application
- class Agent
- class AccessQueryAgent
- class QoTQueryTimer
- class QoTOutQueue
- class TransportQueryTimer
- class StatTimer

B.31.1 Detailed Description

QoT node.

The key component of a QoT node.

B.31.2 Constructor & Destructor Documentation

B.31.2.1 QoTNode::QoTNode ()

Class constructor.

B.31.3 Member Function Documentation

B.31.3.1 void QoTNode::checkCallBack ()

Check callback queue.

The master QoT node stops data transfer when conducting transport availability queries. Events during the query process are inserted to a callback queue. After the availability queries are done, QoT node checks the callback queue to handle those events.

B.31.3.2 QoTOutQueue* QoTNode::checkOutputBuff (nsaddr_t *dest*)

Retrieve the send buffer for a connection.

Parameters:

dest The address of the remote device. This parameter is used to identify the connection.

Returns:

The pointer to the send buffer of the specified connection.

B.31.3.3 void QoTNode::checkRecvBuff (DevTabEntry * *entry*)

Check out session layer data packets stored in the QoT node receive buffer, and move them to the corresponding send buffer.

Parameters:

entry The Remote Device Table entry that corresponds to the connection.

B.31.3.4 void QoTNode::clearDummy () [inline]

Clear the data stored in the dummy_.

B.31.3.5 int QoTNode::command (int *argc*, const char ∗const ∗ *argv*)

Command function that implements Tcl-C++ interfacing functions.

Parameters:

argc Argument count.

argv Argument vector.

B.31.3.6 void QoTNode::connectAccept (sharedT ∗ *share*, nsaddr_t *dst*)

QOT_CONNECT_ACCEPT.

Upon receiving an connection request message, the slave QoT nodes decides whether to accept the request or not. If accept, response with this function.

Parameters:

share The shared transport that the connection is to be established.

dst The address of the remote device

B.31.3.7 void QoTNode::connectReject (Packet ∗ *p*, transport_stack ∗ *stack*, QoTReason *r*)

QOT_CONNECT_REJECT.

Upon receiving an connection request message, the slave QoT nodes decides whether to accept the request or not. If reject, response with this function.

Parameters:

p The connect request packet.

stack The transport stack over which the request was received.

r The reason that the request is rejected.

B.31.3.8 void QoTNode::connectRequest (sharedT * *share*, nsaddr_t *dst*)

QOT_CONNECT_REQUEST.

After the connection establishment phase, the master QoT node requests for connection if there is any available transport shared with the remote node.

Parameters:

share The shared transport over which the QoT node requests the connection to be established.

dst The address of the remote device.

B.31.3.9 void QoTNode::dataSend (DevTabEntry * *entry*)

QOT_DATA_SEND.

Upon receiving the connect accept from the slave node, the master QoT node starts to transmit data packet by this function.

Parameters:

entry The Remote Device Table entry for this connection.

B.31.3.10 void QoTNode::dataSync (DevTabEntry * *entry*)

QOT_DATA_SYNC_POINT.

Once the master node found that the send buffer is full, it requests a data synchronization through this function.

Parameters:

entry The Remote Device Table entry for this connection.

B.31.3.11 void QoTNode::devDiscover (nsaddr_t *dst*)

Discover remote device.

Upon receiving a session layer data packet, the QoT node checks if there exists a connection for the this packet destination. If not, QoT initiates a remote device discovery process to establish a connection for this packet.

Parameters:

dst The address of the remote deice.

B.31.3.12 void QoTNode::disconnectAccept (sharedT * *share*, nsaddr_t *dst*)

QOT_DISCONNECT_ACCEPT.

The slave QoT node accepts the disconnect request.

Parameters:

share The shared transport that this request was received.

dst The address of the remote device.

B.31.3.13 void QoTNode::disconnectRequest (sharedT * *share*, nsaddr_t *dst*)

QOT_DISCONNECT_REQUEST.

The master QoT node requests to disconnect the data transfer.

Parameters:

share The shared transport that this request is sent over.

dst The address of the remote device.

B.31.3.14 void QoTNode::downRecv (int *size*, AppData * *appdata*, const char * *flags* = 0)

Receive a packet from the session layer protocol.

Upon receiving a packet from the session layer, QoT first checks if there already exists a connection for this packet. If so, QoT node directly moves the packet of corresponding send buffer, where the session layer packet is segmented into QoT data packets. If not, QoT node temporary stores it in its receive buffer, and starts establishing the connection.

Parameters:

size The size of the session layer data packet.

appdata Pointer to the AppData.

flags Pakcet flags.

B.31.3.15 transport_stack* QoTNode::get_trans_by_tag (char * *s*)

Retrieve a transport stack according to the provided string.

Parameters:

s String that identifis the type of the transport stack, such as WIFI or BT.

B.31.3.16 nsaddr_t QoTNode::getDest () [inline]

Retrieve the address of the remote device.

B.31.3.17 hdr_qot* QoTNode::getDummy () [inline]

Retrieve the QoT header stored in the dummy_.

B.31.3.18 transport_stack* QoTNode::getStack (char * *tag*)

Retrieve a transport stack of the QoT node.

Parameters:

tag String that identifies the type of the transport, such as WIFI or BT.

Returns:

The pointer to the transport stack of the specified type.

B.31.3.19 void QoTNode::linkBreak (Packet * *p*)

Link break handler.

Each QoT packet includes a pointer to this function. In case of a link break, this function is called.

Parameters:

p The packet that was being sent when the link was broken.

B.31.3.20 void QoTNode::printLocation ()

Print the coordiates of the transports of this QoT node.

B.31.3.21 void QoTNode::printQoTPacket (hdr_qot * *hdr*)

Print QoT packet information.

A QoT helper function. Output the content of a QoT packet to standard output.

Parameters:

hdr The header of the QoT packet that is to be printed.

B.31.3.22 void QoTNode::recv (Packet * *p*, transport_stack * *stack*)

Receive incoming packets.

the main entrance of a QoT node. Handle incoming packets from underlying protocols.

Parameters:

p The pointer to the received packet.

stack The transport stack over which the packet is received.

B.31.3.23 void QoTNode::resumeAccept (DevTabEntry * *entry*, nsaddr_t *dst*)

QOT_RESUME_ACCEPT.

The slave QoT node accepts the resume request.

Parameters:

entry The Remote Device Table entry for this connection.

dst The address of the remote device.

B.31.3.24 void QoTNode::resumeReject (sharedT * *share*, nsaddr_t *dst*)

QOT_RESUME_REJECT.

The slave QoT node rejects the resume request.

Parameters:

entry The Remote Device Table entry for this connection.

dst The address of the remote device.

B.31.3.25 void QoTNode::resumeRequest (DevTabEntry * *entry*, nsaddr_t *dst*)

QOT_RESUME_REQUEST.

After a downgrade transport switching, the master QoT node request to resume the data transfer.

Parameters:

entry The Remote Device Table entry for this connection.

dst The address of the remote device.

B.31.3.26 void QoTNode::sendDown (transport_stack * *s*)

Send a QoT packet down to the underlying protocol.

Parameters:

s The transport stack that is used to send the packet.

B.31.3.27 void QoTNode::setApp (Application * *app*) [inline]

Set the pointer to the session layer protocol.

B.31.3.28 void QoTNode::setDummy (hdr_qot * *hdr*)

Set the dummy_.

B.31.3.29 void QoTNode::setTPM (QTPM * *tpm*) [inline]

Set the pointer to a TPM module.

B.31.3.30 void QoTNode::switchAccept (sharedT * *share*, nsaddr_t *dst*)

QOT_SWITCH_ACCEPT.

The slave QoT node accepts the transport switching request.

Parameters:

share The shared transport that the data traffic is going to be switched to.

dst The address of the remote device.

B.31.3.31 void QoTNode::switchQuery (DevTabEntry * *entry*, SwitchType *type*)

QOT_SWITCH_QUERY.

Before an upgrade transport switching, the master QoT node first query the availability of the desired transport.

Parameters:

entry The Remote Device Table for this connection.

type Transport switching type.

B.31.3.32 void QoTNode::switchQueryResponse (DevTabEntry * *entry*, nsaddr_t *dst*)

QOT_SWITCH_QUERY_RESPONSE.

Upon successfully received a switch query message, the slave QoT node responses by this function.

Parameters:

entry The Remote Device Table for this connection.

dst The address of the remote device.

B.31.3.33 void QoTNode::switchReject (sharedT * *share*, nsaddr_t *dst*)

QOT_SWITCH_REJECT.

The slave QoT node rejects the transport switching request.

Parameters:

share The shared transport over which the request was received.

dst The address of the remote device.

B.31.3.34 void QoTNode::switchRequest (sharedT * *share*, nsaddr_t *dst*)

QOT_SWITCH_REQUEST.

Upon receiving the QOT_SWITCH_QUERY_RESPONSE, the master QoT node requests a transport switching over the new transport.

Parameters:

share The shared transport that the data traffic is going to be switched to.

dst The address of the remote device.

B.31.3.35 void QoTNode::transInfoQuery (sharedT * *share*, nsaddr_t *dst*)

QOT_TRANSPORT_INFO_QUERY.

The master QoT node conducts periodic transport availability queries on all the shared transports.

Parameters:

share The shared transport that is to be queried.

dst The address of the queried transport on the remote device.

B.31.3.36 void QoTNode::transInfoQueryResponse (Packet * *p*, transport_stack * *stack*, double *request*, double *signal*, int *count*)

QOT_TRANSPORT_INFO_QUERY_RESPONSE.

Upon receiving a periodic transport availability query, the slave QoT node sends a response message.

Parameters:

p The received packet.

stack The transport stack that the query message was received.

signal The signal request.

count The sequence number for this query message. Used to examine if the query message has expired.

B.31.3.37 void QoTNode::transQuery (nsaddr_t *dst*, transport_stack * *stack*)

QOT_TRANSPORT_QUERY.

During the process of connection establishment, the master QoT node queries all the supported transport to find out the shared transports with the remote device.

Parameters:

dst The address of the remote device.

stack The transport stack that is to be queried.

B.31.3.38 void QoTNode::transQueryResponse (Packet * *p*, transport_stack * *stack*)

QOT_TRANSPORT_QUERY_RESPONSE.

Upon receiving a transport query message from the master QoT node, the slave QoT node sends a response message.

Parameters:

p The received packet.

stack The transport stack through which the transport query message is received.

B.31.4 Friends And Related Function Documentation

B.31.4.1 friend class AccessQueryAgent [friend]

B.31.4.2 friend class Agent [friend]

B.31.4.3 friend class Application [friend]

B.31.4.4 friend class DevTabEntry [friend]

B.31.4.5 friend class QoTBrain [friend]

B.31.4.6 friend class QoTOutQueue [friend]

B.31.4.7 friend class QoTQueryTimer [friend]

B.31.4.8 friend class StatTimer [friend]

B.31.4.9 friend class TransportQueryTimer [friend]

B.31.5 Member Data Documentation

B.31.5.1 agent_value QoTNode::agent_v [private]

B.31.5.2 Application* QoTNode::app_ [private]

Pointer to the attached session layer protocol.

B.31.5.3 QoTBrain* QoTNode::brain [private]

Pointer to the QoT brain.

B.31.5.4 int QoTNode::busy_

The flag that identifies if the QoT node is performing transport availability query.

B.31.5.5 CallBack* QoTNode::call_back_h [private]

Head of the callback link list.

B.31.5.6 CallBack* QoTNode::call_back_t [private]

Tail of the callback link list.

B.31.5.7 int QoTNode::do_query [private]

B.31.5.8 nsaddr_t QoTNode::dst_ [private]

The address of the remote device.

B.31.5.9 hdr_qot* QoTNode::dummy_ [private]

The buffer that transfers QoT packet header information to the transport agent when it creates a packet.

B.31.5.10 RDT* QoTNode::rd_table [private]

Pointer to the Remote Device Table.

B.31.5.11 QoTQueue* QoTNode::recv_buff [private]

QoT receive buffer.

B.31.5.12 StatTimer* QoTNode::s_timer

Statistical timer.

B.31.5.13 QTPM* QoTNode::tpm_ [private]

Pointer to a TPM module.

B.31.5.14 TransportQueryTimer* QoTNode::tq_timer [private]

Transport query timer.

B.31.5.15 transport_stack* QoTNode::trans_head [private]

Head of the transport stack link list.

B.31.5.16 int QoTNode::transport_nn [private]

The documentation for this class was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.32 QoTOutQueue Class Reference

The send queue within a QoT node.

`#include <qot.h>`

Inherits QoTQueue.

Inheritance diagram for QoTOutQueue:Collaboration diagram for QoTOutQueue:

Public Member Functions

- QoTOutQueue (DevTabEntry *entry)

  Class constructor.

- int enque (app_data *d)

  Enqueue the session layer packets into the queue.

- void send ()

  Send the QoT packets in the send buffer.

- void deque (unsigned id)

  Dequeue the acknowledged packets in the queue.

- void insertSend (hdr_qot *hdr)
- hdr_qot * getNext ()

  Return the next available packet.

- void set_entry (DevTabEntry *entry)

  Set the pointer entry_ to an entry in the Remote Device Table.

- int my_credit ()
- void setCredit (unsigned credit)
- int check_infinite_send ()
- app_data ∗ check_app_buff ()

  Return the pointer to app_buff.

- unsigned getPktID ()

  Return the next available QoT packet ID.

- void syncCheck (unsigned id)

  The synchronization check at the QoT layer of the sending node.

- void requestSync ()

  Request data synchronization.

- void flushQueue ()

Private Member Functions

- hdr_qot ∗ remove_head ()

  Remote queue head.

- int createHdr (int size, app_data ∗data)

  Create a new QoT packet.

- void retransmit ()

  Retransmit the packets in the send buffer.

Private Attributes

- int infinite_send_

  Infinite data available to send.

- app_data ∗ app_buff

  A buffer for app_data.

- DevTabEntry ∗ entry_

  Pointer to the entry that this queue belongs to in the Remote Device Table.

- hdr_qot ∗ head_

  Head of the queue.

- hdr_qot ∗ tail_

  Tail of the queue.

- hdr_qot ∗ send_

  Pointer to the QoT data packet that is ready to be sent next.

- unsigned qot_pkt_id

  QoT data packet id.

- int credit_

Friends

- class DevTabEntry
- class QoTNode

B.32.1 Detailed Description

The send queue within a QoT node.

QoT node creates a send queue for each connection when dealing with reliable transport protocol type, such as TCP.

B.32.2 Constructor & Destructor Documentation

B.32.2.1 QoTOutQueue::QoTOutQueue (DevTabEntry * *entry*)

Class constructor.

B.32.3 Member Function Documentation

B.32.3.1 app_data* QoTOutQueue::check_app_buff () [inline]

Return the pointer to app_buff.

B.32.3.2 int QoTOutQueue::check_infinite_send () [inline]

If the traffic generator is FTP, this flag is set to 1.

B.32.3.3 int QoTOutQueue::createHdr (int *size*, app_data * *data*) [private]

Create a new QoT packet.

In ns-2, data packets are actually created at the transport layer, TCP or UDP. This function virtually creates a new QoT packet in the sense that the packet is not actually

created here. In stead, this function creates a new QoT header structure, and fills corresponding fields. When the data packet is actually created at the transport layer, the QoT header created here is copied to the packet.

Parameters:

size The size of the QoT data packet.

data The app_data buffer that stores the destination address of the session layer data packets.

Returns:

return 1 if the queue is already full, 0 otherwise.

B.32.3.4 void QoTOutQueue::deque (unsigned *id*)

Dequeue the acknowledged packets in the queue.

Parameters:

id The sequence number of the packet that the receiving node last received.

B.32.3.5 int QoTOutQueue::enque (app_data * *d*) [virtual]

Enqueue the session layer packets into the queue.

Session layer packets are segmented to fit the size of QoT packets before enqueue into the send buffer.

Reimplemented from QoTQueue.

B.32.3.6 void QoTOutQueue::flushQueue ()

B.32.3.7 hdr_qot* QoTOutQueue::getNext ()

Return the next available packet.

Return the packet that is pointed by send_ if it is not equal to NULL, otherwise return NULL. If send_ is not equal to NULL, it moves towards tail_ for one packet.

B.32.3.8 unsigned QoTOutQueue::getPktID () [inline]

Return the next available QoT packet ID.

QoT assigns a sequence number to each data packet.

B.32.3.9 void QoTOutQueue::insertSend (hdr_qot * *hdr*)

The inserted packet will be the next available packet to send.

B.32.3.10 int QoTOutQueue::my_credit () [inline]

B.32.3.11 hdr_qot* QoTOutQueue::remove_head () [private]

Remote queue head.

Removes the head of the queue. Returns the original queue head.

Returns:

hdr_qot The QoT data packet that was the head of the queue.

Reimplemented from QoTQueue.

B.32.3.12 void QoTOutQueue::requestSync ()

Request data synchronization.

The QoT layer at the sending node requests a data synchronization once its send buffer is full.

B.32.3.13 void QoTOutQueue::retransmit () [private]

Retransmit the packets in the send buffer.

Retransmit the un-acknowledged and new packets.

B.32.3.14 void QoTOutQueue::send ()

Send the QoT packets in the send buffer.

Once the send buffer is full, the QoT layer on the sending node requests a data synchronization to release the acknowledged packets.

B.32.3.15 void QoTOutQueue::set_entry (DevTabEntry * *entry*) [inline]

Set the pointer entry_ to an entry in the Remote Device Table.

B.32.3.16 void QoTOutQueue::setCredit (unsigned *credit*) [inline]

B.32.3.17 void QoTOutQueue::syncCheck (unsigned *id*)

The synchronization check at the QoT layer of the sending node.

The QoT layer at the sending node requests a data synchronization once its send buffer is full. Upon receiving the synchronization packet from the receiving node, QoT removes acknowledged packets in its send buffer. If new slots are available in the send buffer after the data synchronization, QoT moves the buffer window and continue sending new packets.

Parameters:

id The sequence number of the packet that the QoT layer on the receiving node last received.

B.32.4 Friends And Related Function Documentation

B.32.4.1 friend class DevTabEntry [friend]

B.32.4.2 friend class QoTNode [friend]

B.32.5 Member Data Documentation

B.32.5.1 app_data* QoTOutQueue::app_buff [private]

A buffer for app_data.

This buffer is used to store a app_data information, which is used in case of infinite send of session layer traffic.

B.32.5.2 int QoTOutQueue::credit_ [private]

B.32.5.3 DevTabEntry* QoTOutQueue::entry_ [private]

Pointer to the entry that this queue belongs to in the Remote Device Table.

B.32.5.4 hdr_qot* QoTOutQueue::head_ [private]

Head of the queue.

Reimplemented from QoTQueue.

B.32.5.5 int QoTOutQueue::infinite_send_ [private]

Infinite data available to send.

B.32.5.6 unsigned QoTOutQueue::qot_pkt_id [private]

QoT data packet id.

Starts from 0.

B.32.5.7 hdr_qot* QoTOutQueue::send_ [private]

Pointer to the QoT data packet that is ready to be sent next.

B.32.5.8 hdr_qot* QoTOutQueue::tail_ [private]

Tail of the queue.

Reimplemented from QoTQueue.

The documentation for this class was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.33 QoTPacket Union Reference

Qot packet type.

`#include <hdr.h>`

Collaboration diagram for QoTPacket:

Public Attributes

- qot_con_req con_req

  QOT_CONNECT_REQUEST.

- qot_con_acc con_acc

  QOT_CONNECT_ACCEPT.

- qot_con_rej con_rej

  QOT_CONNECT_REJECT.

- qot_discon_req discon_req

  QOT_DISCONNECT_REQUEST.

- qot_discon_acc discon_acc

  QOT_DISCONNECT_ACCEPT.

- qot_data_snd data_snd

  QOT_DATA_SEND.

- qot_data_sync_pnt data_sync_pnt

QOT_DATA_SYNC_POINT.

- qot_data_sync_req data_sync_req

  QOT_DATA_SYNC_REQUEST.

- qot_swh_qry swh_qry

  QOT_SWITCH_QUERY.

- qot_swh_qry_rep swh_qry_rep

  QOT_SWITCH_QUERY_RESPONSE.

- qot_swh_req swh_req

  QOT_SWITCH_REQUEST.

- qot_swh_acc swh_acc

  QOT_SWITCH_ACCEPT.

- qot_swh_rej swh_rej

  QOT_SWITCH_REJECT.

- qot_rem_req rem_req

  QOT_RESUME_REQUEST.

- qot_rem_acc rem_acc

  QOT_RESUME_ACCEPT.

- qot_rem_rej rem_rej

  QOT_RESUME_REJECT.

- qot_trans_qry trans_qry

  QOT_TRANSPORT_QUERY.

- qot_trans_qry_rep trans_qry_rep

  QOT_TRANSPORT_QUERY_RESPONSE.

- qot_trans_info_qry trans_info_qry

  QOT_TRANSPORT_INFO_QUERY.

- qot_trans_info_qry_rep trans_info_qry_rep

  QOT_TRANSPORT_INFO_QUERY_RESPONSE.

B.33.1 Detailed Description

Qot packet type.

B.33.2 Member Data Documentation

B.33.2.1 qot_con_acc QoTPacket::con_acc

QOT_CONNECT_ACCEPT.

B.33.2.2 qot_con_rej QoTPacket::con_rej

QOT_CONNECT_REJECT.

B.33.2.3 qot_con_req QoTPacket::con_req

QOT_CONNECT_REQUEST.

B.33.2.4 qot_data_snd QoTPacket::data_snd

QOT_DATA_SEND.

B.33.2.5 qot_data_sync_pnt QoTPacket::data_sync_pnt

QOT_DATA_SYNC_POINT.

B.33.2.6 qot_data_sync_req QoTPacket::data_sync_req

QOT_DATA_SYNC_REQUEST.

B.33.2.7 qot_discon_acc QoTPacket::discon_acc

QOT_DISCONNECT_ACCEPT.

B.33.2.8 qot_discon_req QoTPacket::discon_req

QOT_DISCONNECT_REQUEST.

B.33.2.9 qot_rem_acc QoTPacket::rem_acc

QOT_RESUME_ACCEPT.

B.33.2.10 qot_rem_rej QoTPacket::rem_rej

QOT_RESUME_REJECT.

B.33.2.11 qot_rem_req QoTPacket::rem_req

QOT_RESUME_REQUEST.

B.33.2.12 qot_swh_acc QoTPacket::swh_acc

QOT_SWITCH_ACCEPT.

B.33.2.13 qot_swh_qry QoTPacket::swh_qry

QOT_SWITCH_QUERY.

B.33.2.14 qot_swh_qry_rep QoTPacket::swh_qry_rep

QOT_SWITCH_QUERY_RESPONSE.

B.33.2.15 qot_swh_rej QoTPacket::swh_rej

QOT_SWITCH_REJECT.

B.33.2.16 qot_swh_req QoTPacket::swh_req

QOT_SWITCH_REQUEST.

B.33.2.17 qot_trans_info_qry QoTPacket::trans_info_qry

QOT_TRANSPORT_INFO_QUERY.

B.33.2.18 qot_trans_info_qry_rep QoTPacket::trans_info_qry_rep

QOT_TRANSPORT_INFO_QUERY_RESPONSE.

B.33.2.19 qot_trans_qry QoTPacket::trans_qry

QOT_TRANSPORT_QUERY.

B.33.2.20 qot_trans_qry_rep QoTPacket::trans_qry_rep

QOT_TRANSPORT_QUERY_RESPONSE.

The documentation for this union was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.34 QoTQueue Class Reference

the super class for the receive and send buffers within a QoT node.

`#include <qot.h>`

Inherited by QoTOutQueue.

Inheritance diagram for QoTQueue:Collaboration diagram for QoTQueue:

Public Member Functions

- QoTQueue (QoTNode *node)

  Class constructore.

- virtual void enque (app_data *data)

  Enqueue the received session layer packets into the queue.

- virtual app_data * deque (void)

  Dequeue a packets from the queue.

- int removePktWithDst (DevTabEntry *entry)

  Remove packets according to a destination from the queue.

- int len ()

  Returns the length of the queue.

Protected Attributes

- int len_

The length of the queue.

- int limit_

  The maximum length of the queue allowed.

- double timeout_

Private Member Functions

- app_data ∗ remove_head ()

  Removes the head of the queue, returns the original head of the queue.

- void purge (void)
- void verifyQueue (void)

Private Attributes

- QoTNode ∗ node_

  The linakge to the QoT node that this queue belongs to.

- app_data ∗ head_

  The head of the queue.

- app_data ∗ tail_

  The tail of teh queue.

B.34.1 Detailed Description

the super class for the receive and send buffers within a QoT node.

B.34.2 Constructor & Destructor Documentation

B.34.2.1 QoTQueue::QoTQueue (QoTNode * *node*)

Class constructore.

B.34.3 Member Function Documentation

B.34.3.1 virtual app_data* QoTQueue::deque (void) [virtual]

Dequeue a packets from the queue.

B.34.3.2 virtual void QoTQueue::enque (app_data * *data*) [virtual]

Enqueue the received session layer packets into the queue.

Reimplemented in QoTOutQueue.

B.34.3.3 int QoTQueue::len () [inline]

Returns the length of the queue.

B.34.3.4 void QoTQueue::purge (void) [private]

B.34.3.5 app_data* QoTQueue::remove_head () [private]

Removes the head of the queue, returns the original head of the queue.

Reimplemented in QoTOutQueue.

B.34.3.6 int QoTQueue::removePktWithDst (DevTabEntry * *entry*)

Remove packets according to a destination from the queue.

After a QoT node established a connection with a remote node, it removes session layer packets that are destined to the remote node from the receive buffer, and insert the packets into the corresponding send buffer.

Parameters:

entry The entry in the Remote Device Table for this connection. QoT node retrives the destination address from this entry.

B.34.3.7 void QoTQueue::verifyQueue (void) [private]

B.34.4 Member Data Documentation

B.34.4.1 app_data* QoTQueue::head_ [private]

The head of the queue.

Reimplemented in QoTOutQueue.

B.34.4.2 int QoTQueue::len_ [protected]

The length of the queue.

B.34.4.3 int QoTQueue::limit_ [protected]

The maximum length of the queue allowed.

B.34.4.4 QoTNode* QoTQueue::node_ [private]

The linakge to the QoT node that this queue belongs to.

B.34.4.5 app_data* QoTQueue::tail_ [private]

The tail of teh queue.

Reimplemented in QoTOutQueue.

B.34.4.6 double QoTQueue::timeout_ [protected]

The documentation for this class was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.35 QTPM Class Reference

QoT TPM module.

`#include <qot.h>`

Collaboration diagram for QTPM:

Public Member Functions

- QTPM ()

  Class constructor.

- virtual void sendmsg (int sz, AppData ∗d, const char ∗flags=0)

  Send message down.

- virtual void sendmsg (int nbytes, const char ∗flags=0)

  Send message down.

- void setNode (QoTNode ∗node)

  Set pointer to the attached QoT node.

Public Attributes

- int i

Protected Member Functions

- int command (int argc, const char ∗const ∗argv)

Tcl-C++ interfacing function.

Private Attributes

- QoTNode ∗ node_

  Pointer to the QoT node that this TPM is attached to.

B.35.1 Detailed Description

QoT TPM module.

B.35.2 Constructor & Destructor Documentation

B.35.2.1 QTPM::QTPM ()

Class constructor.

B.35.3 Member Function Documentation

B.35.3.1 int QTPM::command (int *argc*, const char ∗const ∗ *argv*) [protected]

Tcl-C++ interfacing function.

B.35.3.2 virtual void QTPM::sendmsg (int *nbytes*, const char * *flags* = 0) [virtual]

Send message down.

This function is used by application layer to send data down.

Parameters:

nbytes The size of the packet.

flags Flags

B.35.3.3 virtual void QTPM::sendmsg (int *sz*, AppData * *d*, const char * *flags* = 0) [virtual]

Send message down.

This function is used by application layer to send data down.

Parameters:

sz The size of the packet.

d AppData

flags Flags

B.35.3.4 void QTPM::setNode (QoTNode * *node*) [inline]

Set pointer to the attached QoT node.

B.35.4 Member Data Documentation

B.35.4.1 int QTPM::i

B.35.4.2 QoTNode* QTPM::node_ [private]

Pointer to the QoT node that this TPM is attached to.

The documentation for this class was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.36 RDT Class Reference

Remote Device Table.

`#include <qot.h>`

Collaboration diagram for RDT:

Public Member Functions

- RDT ()

  Class constructor.

- void setnode (QoTNode *node)

  Set the pointer to the QoT node that this table belongs to.

- QoTNode * node ()

  Retrieve the pointer to the QoT node that this table belongs to.

- DevTabEntry * getTabEntry (nsaddr_t id)

  Retrieve the table entry identified by the remote device ID.

- DevTabEntry * checkTabEntry (nsaddr_t id)

  Check if there exists a table entry with the specified remote device ID.

- int removeTabEntry (nsaddr_t id)

  Remove the table entry from the link list.

- QoTBrain * getBrain ()

Return the pointer to the QoTBrain of the QoT node.

Private Member Functions

- DevTabEntry ∗ createTabEntry (nsaddr_t id)

  Create a table entry with the specified remote address in the table.

Private Attributes

- DevTabEntry ∗ head_

  Head of the table entry link list.

- DevTabEntry ∗ tail_

  Tail of the table entry link list.

- QoTNode ∗ node_

  Pointer to the QoT node that this table belongs to.

- QoTBrain ∗ brain_

  Pointer to the QoTBrain of the QoT node.

Friends

- class QoTNode

B.36.1 Detailed Description

Remote Device Table.

B.36.2 Constructor & Destructor Documentation

B.36.2.1 RDT::RDT () [inline]

Class constructor.

B.36.3 Member Function Documentation

B.36.3.1 DevTabEntry* RDT::checkTabEntry (nsaddr_t *id*)

Check if there exists a table entry with the specified remote device ID.

B.36.3.2 DevTabEntry* RDT::createTabEntry (nsaddr_t *id*) [private]

Create a table entry with the specified remote address in the table.

Parameters:

id The address of the remote device of the connection.

Returns:

The handler to the table entry with the specified remote device address.

B.36.3.3 QoTBrain* RDT::getBrain () [inline]

Return the pointer to the QoTBrain of the QoT node.

B.36.3.4 DevTabEntry* RDT::getTabEntry (nsaddr_t *id*)

Retrieve the table entry identified by the remote device ID.

Return the table entry identified by the remote device ID. If no such entry exists, create one.

Parameters:

id The address and port number of the remote device.

Returns:

The pointer to the table entry identified by the specified ID.

B.36.3.5 QoTNode* RDT::node () `[inline]`

Retrieve the pointer to the QoT node that this table belongs to.

B.36.3.6 int RDT::removeTabEntry (nsaddr_t *id*)

Remove the table entry from the link list.

Parameters:

id The address and port number of the remote device.

Returns:

0 for successful removal, 1 otherwise.

B.36.3.7 void RDT::setnode (QoTNode * *node*) `[inline]`

Set the pointer to the QoT node that this table belongs to.

B.36.4 Friends And Related Function Documentation

B.36.4.1 friend class QoTNode [friend]

B.36.5 Member Data Documentation

B.36.5.1 QoTBrain* RDT::brain_ [private]

Pointer to the QoTBrain of the QoT node.

B.36.5.2 DevTabEntry* RDT::head_ [private]

Head of the table entry link list.

B.36.5.3 QoTNode* RDT::node_ [private]

Pointer to the QoT node that this table belongs to.

B.36.5.4 DevTabEntry* RDT::tail_ [private]

Tail of the table entry link list.

The documentation for this class was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.37 sharedT Struct Reference

Shared transport between two communicating QoT nodes.

`#include <qot.h>`

Collaboration diagram for sharedT:

Public Member Functions

- int getCount ()

  Return the value of the current query sequency number.

- void incrCount ()

  Increment the value of the query sequency number by 1.

Public Attributes

- char * tag_

  The type of this transport, such as WIFI, BT, WUSB or ZIGBEE.

- transport_stack * stack_

  The pointer to the actual transport stack.

- ns_addr_t dest_

  The address and the port number of the same type of transport on the receiving node.

- u_char qresults_
- double query_interval_

The query interval for this transport.

- double power_consumption_

  The power consumption of this transport, in watts.

- double radius_

  The radius of the signal transmission of this transport.

- double data_rate_

  The data rate that this transport can support.

- double utility_
- double throughput

  The measured effective throughput of this transport.

- double delay

  The measured delay on packets transmission.

- double jitter

  The jitter of packets tranfer.

- double signal

  The signal strength when sending packets of this transport.

- int status_
- int p_query
- int query_count

  the sequence number of the transport availablility query.

- QueryTimer ∗ qtimer
- DevTabEntry ∗ entry_

  Linkage to the entry of the Remote Device Table.

- AccessQueryAgent ∗ agent

  Linkage to the access query agent.

- sharedT ∗ prev_

  Linkage to the previous shared transport.

- sharedT ∗ next_

  Linkage to the next shared transport.

B.37.1 Detailed Description

Shared transport between two communicating QoT nodes.

This struct records information of a shared transport between two communicating QoT nodes.

B.37.2 Member Function Documentation

B.37.2.1 int sharedT::getCount () [inline]

Return the value of the current query sequency number.

B.37.2.2 void sharedT::incrCount () [inline]

Increment the value of the query sequency number by 1.

B.37.3 Member Data Documentation

B.37.3.1 AccessQueryAgent* sharedT::agent

Linkage to the access query agent.

An access query agent is responsible for the decision making related issues of a QoT connection, such as predicting the availability of a shared transport and calculating the dynamic transport query intervals.

B.37.3.2 double sharedT::data_rate_

The data rate that this transport can support.

B.37.3.3 double sharedT::delay

The measured delay on packets transmission.

B.37.3.4 ns_addr_t sharedT::dest_

The address and the port number of the same type of transport on the receiving node.

B.37.3.5 DevTabEntry* sharedT::entry_

Linkage to the entry of the Remote Device Table.

The QoT node establishes an entry in the Remote Device Table for each connection. An Entry records information of the shared transports for this connection.

B.37.3.6 double sharedT::jitter

The jitter of packets tranfer.

B.37.3.7 sharedT* sharedT::next_

Linkage to the next shared transport.

B.37.3.8 int sharedT::p_query

B.37.3.9 double sharedT::power_consumption_

The power consumption of this transport, in watts.

B.37.3.10 sharedT* sharedT::prev_

Linkage to the previous shared transport.

The shared transports of this QoT connection are linked together under the corresponding entry in the Remote Device Table.

B.37.3.11 u_char sharedT::qresults_

B.37.3.12 QueryTimer* sharedT::qtimer

B.37.3.13 int sharedT::query_count

the sequence number of the transport availablility query.

This sequence number is used to exam if a received transport availability query has been expired or not.

B.37.3.14 double sharedT::query_interval_

The query interval for this transport.

B.37.3.15 double sharedT::radius_

The radius of the signal transmission of this transport.

B.37.3.16 double sharedT::signal

The signal strength when sending packets of this transport.

B.37.3.17 transport_stack* sharedT::stack_

The pointer to the actual transport stack.

B.37.3.18 int sharedT::status_

B.37.3.19 char* sharedT::tag_

The type of this transport, such as WIFI, BT, WUSB or ZIGBEE.

B.37.3.20 double sharedT::throughput

The measured effective throughput of this transport.

B.37.3.21 double sharedT::utility_

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.38 stack_bt Struct Reference

Pointers to transport stacks of Bluetooth.

`#include <qot.h>`

Public Attributes

- LL ∗ ll
- BNEP ∗ bnep
- L2CAP ∗ l2cap
- LMP ∗ lmp
- Baseband ∗ bb

B.38.1 Detailed Description

Pointers to transport stacks of Bluetooth.

B.38.2 Member Data Documentation

B.38.2.1 Baseband∗ stack_bt::bb

B.38.2.2 BNEP∗ stack_bt::bnep

B.38.2.3 L2CAP* stack_bt::l2cap

B.38.2.4 LL* stack_bt::ll

B.38.2.5 LMP* stack_bt::lmp

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.39 stack_wifi Struct Reference

Pointers to transport stacks of WiFi.

`#include <qot.h>`

Public Attributes

- LL * ll
- PriQueue * ifq
- WirelessChannel * channel

B.39.1 Detailed Description

Pointers to transport stacks of WiFi.

B.39.2 Member Data Documentation

B.39.2.1 WirelessChannel* stack_wifi::channel

B.39.2.2 PriQueue* stack_wifi::ifq

B.39.2.3 LL* stack_wifi::ll

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.40 stack_wusb Struct Reference

Pointers to transport stacks of WUSB.

`#include <qot.h>`

Public Attributes

- LL ∗ ll
- PriQueue ∗ ifq
- WirelessChannel ∗ channel

B.40.1 Detailed Description

Pointers to transport stacks of WUSB.

B.40.2 Member Data Documentation

B.40.2.1 WirelessChannel∗ stack_wusb::channel

B.40.2.2 PriQueue∗ stack_wusb::ifq

B.40.2.3 LL∗ stack_wusb::ll

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.41 stack_zigbee Struct Reference

Pointers to transport stacks of ZigBee.

`#include <qot.h>`

Public Attributes

- LL * ll
- PriQueue * ifq
- WirelessChannel * channel

B.41.1 Detailed Description

Pointers to transport stacks of ZigBee.

B.41.2 Member Data Documentation

B.41.2.1 WirelessChannel* stack_zigbee::channel

B.41.2.2 PriQueue* stack_zigbee::ifq

B.41.2.3 LL* stack_zigbee::ll

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.42 StatTimer Class Reference

Statistical timer.

`#include <qot.h>`

Collaboration diagram for StatTimer:

Public Member Functions

- StatTimer (QoTNode *n)

  Class constructor.

- void printThroughput ()

  Output the throughput statistical results to a file.

- void printPower ()

  Output the power consumption statistical results to a file.

Protected Member Functions

- void expire (Event *e)

Protected Attributes

- QoTNode * node

  Pointer to the QoT node that this timer belongs to.

- double interval

The time interval that this timer collects data.

- int t

  A temporary variable to save cumulative throughput.

- double p

  A temporary variable to save cumulative power consumption.

- char ∗ tag
- throughput ∗ t_head

  Link list head for collected throughput data.

- throughput ∗ t_tail

  Link list tail for collected throughput data.

- p_consumption ∗ p_head

  Link list head for collected power consumption data.

- p_consumption ∗ p_tail

  Link list tail for collected power consumption data.

Friends

- class QoTNode

B.42.1 Detailed Description

Statistical timer.

Collect the throughput and the power consumption over time at the QoT layer.

B.42.2 Constructor & Destructor Documentation

B.42.2.1 StatTimer::StatTimer (QoTNode * *n*) [inline]

Class constructor.

Parameters:

n Pointer to the QoT node that this timer belongs to.

B.42.3 Member Function Documentation

B.42.3.1 void StatTimer::expire (Event * *e*) [protected]

Parameters:

e Event handler.

B.42.3.2 void StatTimer::printPower ()

Output the power consumption statistical results to a file.

B.42.3.3 void StatTimer::printThroughput ()

Output the throughput statistical results to a file.

B.42.4 Friends And Related Function Documentation

B.42.4.1 friend class QoTNode [friend]

B.42.5 Member Data Documentation

B.42.5.1 double StatTimer::interval [protected]

The time interval that this timer collects data.

B.42.5.2 QoTNode* StatTimer::node [protected]

Pointer to the QoT node that this timer belongs to.

B.42.5.3 double StatTimer::p [protected]

A temporary variable to save cumulative power consumption.

B.42.5.4 p_consumption* StatTimer::p_head [protected]

Link list head for collected power consumption data.

B.42.5.5 p_consumption* StatTimer::p_tail [protected]

Link list tail for collected power consumption data.

B.42.5.6 int StatTimer::t [protected]

A temporary variable to save cumulative throughput.

B.42.5.7 throughput* StatTimer::t_head [protected]

Link list head for collected throughput data.

B.42.5.8 throughput* StatTimer::t_tail [protected]

Link list tail for collected throughput data.

B.42.5.9 char* StatTimer::tag [protected]

The documentation for this class was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.43 TAM Class Reference

QoT TAM module.

`#include <qot.h>`

Collaboration diagram for TAM:

Public Member Functions

- TAM ()

  Class constructor.

- void recv (int nbytes, Packet ∗p)

  Receive packets from the underlying transport layer.

- transport_stack ∗ transport ()

  Retrieve the pointer to the attached transport.

- void setTran (transport_stack ∗t)

  Set the pointer to the the attached transport.

- void setNode (QoTNode ∗node)

  Set the pointer to the attached QoT node.

- hdr_qot ∗ get_pkt_hdr ()

  Retrieve the QoT header stored in the dummy_ of the QoT node.

- void clearDummy ()

Clear the dummy_ field in the QoT node.

Private Attributes

- QoTNode ∗ node_

  Pointer to the attached QoT node.

- transport_stack ∗ tran

  Pointer to the underlying transport agent.

B.43.1 Detailed Description

QoT TAM module.

B.43.2 Constructor & Destructor Documentation

B.43.2.1 TAM::TAM ()

Class constructor.

B.43.3 Member Function Documentation

B.43.3.1 void TAM::clearDummy () [inline]

Clear the dummy_ field in the QoT node.

B.43.3.2 hdr_qot* TAM::get_pkt_hdr () [inline]

Retrieve the QoT header stored in the dummy_ of the QoT node.

B.43.3.3 void TAM::recv (int *nbytes*, Packet * *p*)

Receive packets from the underlying transport layer.

Parameters:

nbytes The size of the received packet.

p The pointer to the received packet.

B.43.3.4 void TAM::setNode (QoTNode * *node*) [inline]

Set the pointer to the attached QoT node.

B.43.3.5 void TAM::setTran (transport_stack * *t*) [inline]

Set the pointer to the the attached transport.

B.43.3.6 transport_stack* TAM::transport () [inline]

Retrieve the pointer to the attached transport.

B.43.4 Member Data Documentation

B.43.4.1 QoTNode* TAM::node_ [private]

Pointer to the attached QoT node.

B.43.4.2 transport_stack* TAM::tran [private]

Pointer to the underlying transport agent.

The documentation for this class was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.44 throughput Struct Reference

The throughput link list item used in the statistical timer.

`#include <qot.h>`

Collaboration diagram for throughput:

Public Attributes

- throughput * next

  Pointer to the next item in the link list.

- double t_value

  Throughput value.

- double t

  The time period that the collected data correspond to.

B.44.1 Detailed Description

The throughput link list item used in the statistical timer.

B.44.2 Member Data Documentation

B.44.2.1 throughput* throughput::next

Pointer to the next item in the link list.

B.44.2.2 double throughput::t

The time period that the collected data correspond to.

B.44.2.3 double throughput::t_value

Throughput value.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.45 trans_info Struct Reference

Structure used in QOT_TRANSPORT_QUERY_RESPONSE.

`#include <hdr.h>`

Collaboration diagram for trans_info:

Public Attributes

- char * tag

  Transport type.

- ns_addr_t me

  Transport address and port.

- trans_info * next

B.45.1 Detailed Description

Structure used in QOT_TRANSPORT_QUERY_RESPONSE.

B.45.2 Member Data Documentation

B.45.2.1 ns_addr_t trans_info::me

Transport address and port.

B.45.2.2 trans_info* trans_info::next

B.45.2.3 char* trans_info::tag

Transport type.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/hdr.h

B.46 transport_stack Struct Reference

The protocol stack layer of a transport of a QoT node.

`#include <qot.h>`

Collaboration diagram for transport_stack:

Public Attributes

- char * transport_tag

  Transport type tags, such as WIFI, BT, WUSB, or WUSB.

- char * transport_t

  Transport layer protocol type, such as TCP or UDP.

- char * routing_t

  Routing layer protocol type, such as AODV or DSR.

- Agent * transport_agent

  The pointer to the transport layer agent.

- Agent * rt_agent

  The pointer to the routing layer agent.

- Node * node

  The pointer to a transport of this QoT node.

- qot_stack stack

A union that contains pointers to protocol layers that are below the routing layer of a transport.

- ns_addr_t me

  The address and the port number of this transport.

- transport_stack ∗ next

  Linkage to the next transport of the transport link list within a QoT node.

- transport_stack ∗ prev

  Linkage to the previous transport of the transport link list within a QoT ndoe.

B.46.1 Detailed Description

The protocol stack layer of a transport of a QoT node.

In ns-2, the structures of transports maybe significantly different. A union, stack, is used to present layers that are below the routing layer.

B.46.2 Member Data Documentation

B.46.2.1 ns_addr_t transport_stack::me

The address and the port number of this transport.

B.46.2.2 transport_stack∗ transport_stack::next

Linkage to the next transport of the transport link list within a QoT node.

All transports of a QoT node are linked together.

B.46.2.3 Node* transport_stack::node

The pointer to a transport of this QoT node.

A transport of a QoT node is equivalent to a homogeneous node in ns-2.

B.46.2.4 transport_stack* transport_stack::prev

Linkage to the previous transport of the transport link list within a QoT ndoe.

B.46.2.5 char* transport_stack::routing_t

Routing layer protocol type, such as AODV or DSR.

B.46.2.6 Agent* transport_stack::rt_agent

The pointer to the routing layer agent.

B.46.2.7 qot_stack transport_stack::stack

A union that contains pointers to protocol layers that are below the routing layer of a transport.

Protocol layer structures that are below the routing layer may be significantly differnt between transports of a QoT node.

B.46.2.8 Agent* transport_stack::transport_agent

The pointer to the transport layer agent.

B.46.2.9 char* transport_stack::transport_t

Transport layer protocol type, such as TCP or UDP.

B.46.2.10 char* transport_stack::transport_tag

Transport type tags, such as WIFI, BT, WUSB, or WUSB.

The documentation for this struct was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.47 TransportQueryTimer Class Reference

Transport query timer.

`#include <qot.h>`

Collaboration diagram for TransportQueryTimer:

Public Member Functions

- TransportQueryTimer (QoTNode ∗n)
- void setStack (transport_stack ∗s)

  Set the pointer to a transport stack.

- void setID (nsaddr_t i)

  Set remote device ID.

- void setEntry (DevTabEntry ∗e)

  Set the pointer to a Remote Device Table entry.

Protected Member Functions

- virtual void expire (Event ∗e)

Protected Attributes

- QoTNode ∗ node

  Pointer to the QoT node that this timer belongs to.

- transport_stack ∗ stack

Pointer to a transport stack.

- DevTabEntry * entry

  Pointer to a Remote Device Table entry.

- nsaddr_t remote_id

  Address of the remote device.

B.47.1 Detailed Description

Transport query timer.

When establishing a connection with the remote device, the master QoT node queries all the supported transports to find out the shared transports with the remote device. If the master QoT node doesn't receive any response over a queried transport before this timer expires, it marks this transport as unavailable.

B.47.2 Constructor & Destructor Documentation

B.47.2.1 TransportQueryTimer::TransportQueryTimer (QoTNode * *n*) [inline]

Parameters:

n Pointer to the QoT node that this timer belongs to.

B.47.3 Member Function Documentation

B.47.3.1 virtual void TransportQueryTimer::expire (Event * *e*) [protected, virtual]

Parameters:

e Event handler.

B.47.3.2 void TransportQueryTimer::setEntry (DevTabEntry * *e*) [inline]

Set the pointer to a Remote Device Table entry.

B.47.3.3 void TransportQueryTimer::setID (nsaddr_t *i*) [inline]

Set remote device ID.

B.47.3.4 void TransportQueryTimer::setStack (transport_stack * *s*) [inline]

Set the pointer to a transport stack.

B.47.4 Member Data Documentation

B.47.4.1 DevTabEntry* TransportQueryTimer::entry [protected]

Pointer to a Remote Device Table entry.

B.47.4.2 QoTNode* TransportQueryTimer::node [protected]

Pointer to the QoT node that this timer belongs to.

B.47.4.3 nsaddr_t TransportQueryTimer::remote_id [protected]

Address of the remote device.

B.47.4.4 transport_stack* TransportQueryTimer::stack [protected]

Pointer to a transport stack.

The documentation for this class was generated from the following file:

- /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h

B.48 /ns-2.28/qot/hdr.h File Reference

```
#include "ns-process.h"
```

Include dependency graph for hdr.h:

Classes

- struct qot_con_req

  QOT_CONNECT_REQUEST message.

- struct qot_con_acc

  QOT_CONNECT_ACCEPT message.

- struct qot_con_rej

  QOT_CONNECT_REJECT message.

- struct qot_discon_req

  QOT_DISCONNECT_REQUEST message.

- struct qot_discon_acc

  QOT_DISCONNECT_ACCEPT message.

- struct qot_data_snd

  QOT_DATA_SEND message.

- struct qot_data_sync_pnt

  QOT_DATA_SYNC_POINT message.

- struct qot_data_sync_req

  QOT_DATA_SYNC_REQUEST message.

- struct qot_swh_qry

  QOT_SWITCH_QUERY message.

- struct prio_info

  Transport priority information.

- struct qot_swh_qry_rep

  QOT_SWITCH_QUERY_RESPONSE.

- struct qot_swh_req

  QOT_SWITCH_REQUEST.

- struct qot_swh_acc

  QOT_SWITCH_ACCEPT message.

- struct qot_swh_rej

  QOT_SWITCH_REJECT message.

- struct qot_rem_req

  QOT_RESUME_REQUEST message.

- struct qot_rem_acc

  QOT_RESUME_ACCEPT messsage.

- struct qot_rem_rej

  QOT_RESUME_REJECT message.

- struct qot_trans_qry

  QOT_TRANSPORT_QUERY message.

- struct trans_info

  Structure used in QOT_TRANSPORT_QUERY_RESPONSE.

- struct qot_trans_qry_rep

  QOT_TRANSPORT_QUERY_RESPONSE messge content.

- struct qot_trans_info_qry

  QOT_TRANSPORT_INFO_QUERY messge content.

- struct qot_trans_info_qry_rep

  QOT_TRANSPORT_INFO_QUERY_RESPONSE messge content.

- union QoTPacket

  Qot packet type.

Enumerations

- enum QoTPacketType {

  QOT_CONNECT_REQUEST, QOT_CONNECT_ACCEPT, QOT_CONNECT_REJECT, QOT_DISCONNECT_REQUEST,

  QOT_DISCONNECT_ACCEPT, QOT_DATA_SEND, QOT_DATA_SYNC_POINT, QOT_DATA_SYNC_REQUEST,

QOT_SWITCH_QUERY, QOT_SWITCH_QUERY_RESPONSE, QOT_SWITCH_REQUEST, QOT_SWITCH_ACCEPT,

QOT_SWITCH_REJECT, QOT_RESUME_REQUEST, QOT_RESUME_ACCEPT, QOT_RESUME_REJECT,

QOT_TRANSPORT_QUERY, QOT_TRANSPORT_QUERY_RESPONSE, QOT_TRANSPORT_INFO_QUERY, QOT_TRANSPORT_INFO_QUERY_RESPONSE,

INVALID }

An enumeration of possible QoT states.

- enum QoTReason { INVALID_NODE, INVALID_TRANSPORT, LOW_POWER, UNINITIALIZED }

  An enumeration of possible reasons that a QoT node rejects a connect request.

- enum SwitchType { UPGRADE, DOWNGRADE }

  Transport switching type.

- enum ApplicationType {

  VoIP, VIDEO, STREAM, MAIL,

  FTP, HTTP }

  Application type.

B.48.1 Enumeration Type Documentation

B.48.1.1 enum ApplicationType

Application type.

Enumerator:

VoIP

VIDEO

STREAM

MAIL

FTP

HTTP

B.48.1.2 enum QoTPacketType

An enumeration of possible QoT states.

Enumerator:

QOT_CONNECT_REQUEST

QOT_CONNECT_ACCEPT

QOT_CONNECT_REJECT

QOT_DISCONNECT_REQUEST

QOT_DISCONNECT_ACCEPT

QOT_DATA_SEND

QOT_DATA_SYNC_POINT

QOT_DATA_SYNC_REQUEST

QOT_SWITCH_QUERY

QOT_SWITCH_QUERY_RESPONSE

QOT_SWITCH_REQUEST

QOT_SWITCH_ACCEPT

QOT_SWITCH_REJECT

QOT_RESUME_REQUEST

QOT_RESUME_ACCEPT

QOT_RESUME_REJECT

QOT_TRANSPORT_QUERY

QOT_TRANSPORT_QUERY_RESPONSE

QOT_TRANSPORT_INFO_QUERY

QOT_TRANSPORT_INFO_QUERY_RESPONSE

INVALID

B.48.1.3 enum QoTReason

An enumeration of possible reasons that a QoT node rejects a connect request.

Enumerator:

INVALID_NODE

INVALID_TRANSPORT

LOW_POWER

UNINITIALIZED

B.48.1.4 enum SwitchType

Transport switching type.

Enumerator:

UPGRADE

DOWNGRADE

B.49 /Users/lei/ns/ns-allinone-2.28/qot.ns-2.28/thesis/qot.h File Reference

```
#include <assert.h>
#include "hdr_qot.h"
#include "bi-connector.h"
#include "node.h"
#include "mobilenode.h"
#include "bt-node.h"
#include "ll.h"
#include "mac-timers.h"
#include "object.h"
#include "agent.h"
#include "priqueue.h"
#include "app.h"
```

Include dependency graph for qot.h:

Classes

- struct stack_wifi

  Pointers to transport stacks of WiFi.

- struct stack_bt

  Pointers to transport stacks of Bluetooth.

- struct stack_zigbee

Pointers to transport stacks of ZigBee.

- struct stack_wusb

  Pointers to transport stacks of WUSB.

- union qot_stack

  The union of possible transport stacks of a QoT node.

- struct transport_stack

  The protocol stack layer of a transport of a QoT node.

- struct app_data

  The application data that a session layer protocol sends down to the QoT layer.

- struct sharedT

  Shared transport between two communicating QoT nodes.

- class QoTQueue

  the super class for the receive and send buffers within a QoT node.

- class QoTOutQueue

  The send queue within a QoT node.

- struct act_trans_list

  Active transport link list.

- class DevTabEntry

  Entry in the Remote Device Table.

- class RDT

  Remote Device Table.

- class QoTBrain

  QoT Brain.

- struct CallBack

  Callback link list entry.

- class TransportQueryTimer

  Transport query timer.

- struct throughput

  The throughput link list item used in the statistical timer.

- struct p_consumption

  The power consumption link list item used in the statistical timer.

- class StatTimer

  Statistical timer.

- class QoTNode

  QoT node.

- class QTPM

  QoT TPM module.

- class TAM

  QoT TAM module.

Defines

- #define QOT_BROADCAST ((u_int32_t) 0xffffffff)
- #define MAX_RECV_BUFFER 100
- #define BUFFER_MAX_LEN 64
- #define MAXIMUM_BURST 10
- #define QOT_BUFFER_TIMEOUT 30
- #define QOT_PKT_LEN (512 - 3 - sizeof(qot_data_snd))
- #define CONNECT_TIMEOUT 3
- #define DATA_TIMEOUT 3
- #define RESUME_TIMEOUT 3
- #define SWITCH_TIMEOUT 3
- #define QUERY_TIMEOUT 0.5
- #define TRANSPORT_QUERY_TIMEOUT 2

Enumerations

- enum QoTState {

  QOT_READY, QOT_CONNECTING, QOT_CONNECTED, QOT_DISCONNECTING,

  QOT_DISCONNECTED, QOT_DATA_FULL, QOT_TRANS_SWITCH, QOT_TRANS_DISC,

  QOT_PRIMARY_DROPPED, QOT_SECONDARY_DROPPED }

  QoT State.

B.49.1 Define Documentation

B.49.1.1 #define BUFFER_MAX_LEN 64

B.49.1.2 #define CONNECT_TIMEOUT 3

B.49.1.3 #define DATA_TIMEOUT 3

B.49.1.4 #define MAX_RECV_BUFFER 100

B.49.1.5 #define MAXIMUM_BURST 10

B.49.1.6 #define QOT_BROADCAST ((u_int32_t) 0xffffffff)

B.49.1.7 #define QOT_BUFFER_TIMEOUT 30

B.49.1.8 #define QOT_PKT_LEN (512 - 3 - sizeof(qot_data_snd))

B.49.1.9 #define QUERY_TIMEOUT 0.5

B.49.1.10 #define RESUME_TIMEOUT 3

B.49.1.11 #define SWITCH_TIMEOUT 3

B.49.1.12 #define TRANSPORT_QUERY_TIMEOUT 2

B.49.2 Enumeration Type Documentation

B.49.2.1 enum QoTState

QoT State.

Enumeration of QoT states as described in QoT specification.

Enumerator:

QOT_READY

QOT_CONNECTING

QOT_CONNECTED

QOT_DISCONNECTING

QOT_DISCONNECTED

QOT_DATA_FULL

QOT_TRANS_SWITCH

QOT_TRANS_DISC

QOT_PRIMARY_DROPPED

QOT_SECONDARY_DROPPED

Bibliography

[1] M. Stemm and R. H. Katz, "Vertical handoffs in wireless overlay networks," *Mobile Networks and Applications*, vol. 4, 1999.

[2] S.-E. Kim and J. Copeland, "Tcp for seamless vertical handoff in hybrid mobile data networks," in *Global Telecommunications Conference*. IEEE, 2003.

[3] M. Baker, X. Zhao, and J. Stone, "Supporting mobility in mosquitonet," in *proceedings of the 1996 USENIX Technical Conference, San Diego, California*, 1996.

[4] A. Salkintzis, C. Fors, and R. Pazhyannur, "Wlan-gprs integration for next-generation mobile data networks," in *Wireless Communications*. IEEE, 2002.

[5] M. Buddhikot, G. Chandranmenon, S. Han, Y. Lee, S. Miller, and L. Salgarelli, "Integration of 802.11 and third-generation wireless data networks," in *INFOCOM*. IEEE, 2003.

[6] S. G. et. al., "Demonstrating seamless handover of multi-hop networks," in *REALMAN '06: Proceedings of the second international workshop on Multi-hop ad hoc networks: from theory to reality*. New York, NY, USA: ACM Press, 2006.

[7] "Ieee 802.21 working group, media independent handover," http://www.ieee802.org/21.

[8] M. G. Williams, "Directions in media independent handover," *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, vol. Vol.E88-A, no. 7, pp. 1772–1776, 2005.

[9] C. D. Knutson, R. W. Woodings, S. B. Barnes, H. R. Duffin, and J. M. Brown, "Dynamic autonomous transport selection in heterogeneous wireless environments,"

in *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, 2004.

[10] R. W. Woodings, D. Joos, T. Clifton, and C. D. Knutson, "Rapid heterogeneous ad hoc connection establishment: Accelerating bluetooth inquiry using irda," in *Proceedings of the Third Annual IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2002.

[11] J. C. Funk, H. R. Duffin, L. Dai, and C. D. Knutson, "Inverse multiplexing in short-range multi-transport wireless communications," in *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC), New Orleans, Louisiana*, Mar. 17-19, 2003.

[12] M. P. et. al., "Ria: An rf interference avoidance algorithm for heterogeneous wireless networks," will be submitted for WCNC 2007.