# SAND REPORT

SAND2004-6232
UNLIMITED RELEASE
Printed November 2004

# Modular Architecture for Sensor Systems (MASS): Description, Analysis, Simulation, and Implementation

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL85000



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.



Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

# Modular Architecture for Sensor Systems (MASS): Description, Analysis, Simulation, and Implementation

Jesse Davis, Doug Stark, Nick Edmonds

Sandia National Laboratories
MS9159 P.O. Box 969
Livermore, CA 94551-0969

## ABSTRACT

A particular engineering aspect of distributed sensor networks that has not received adequate attention is the system level hardware architecture of the individual nodes of the network. A novel hardware architecture based on an idea of task specific modular computing is proposed to provide for both the high flexibility and low power consumption required for distributed sensing solutions. The power consumption of the architecture is mathematically analyzed against a traditional approach, and guidelines are developed for application scenarios that would benefit from using this new design. Furthermore a method of decentralized control for the modular system is developed and analyzed. Finally, a few policies for power minimization in the decentralized system are proposed and analyzed.

## Table of Contents:

**Table of Figures:**

**Table of Tables:**

# INTRODUCTION

Much of the research focus in the area of distributed sensor networks has been devoted to developing power aware or power conservative wireless networking, operating systems, or application software. The hardware aspects of these systems have been largely overlooked, yet many potential benefits may be derived by considering the node system as a whole including both software and hardware components. Furthermore, implementation of distributed sensor networks can be highly application dependent, and this realization provides a new systems perspective for engineering generalized solutions.

Only two system level design approaches have previously been made regarding node hardware. One approach has been to develop highly optimized, low power, and inflexible systems that are specific to a single application. The other approach has been to engineer non-optimal, high power, yet extremely flexible systems that can be adapted to many different applications. These approaches are unsatisfactory to produce programmatically efficient, deployable sensor systems. The inflexible systems become one time solutions that cannot be adapted and are thus programmatically expensive, yet the flexible systems consume large amounts of power and resources and hence require frequent attendance once deployed. The remainder of this document will discuss and analyze an architecture developed to strike a balance between the flexibility, optimality, and power consumption of these two previous approaches. The architecture is specifically designed to satisfy the needs of event-driven sensor networks.

## PROPOSED NODE ARCHITECTURE

There are several key features that an ideal sensor network platform would have. It would be low power, robust against failure, extensible and adaptable to many applications, upgradeable as components are advanced, and have capabilities for complex computational requirements. These features are primary motivations for the idea to develop a task separated, modular, decentralized architecture. The modularity will immediately allow better extensibility and upgradeability in comparison to more traditional centralized approaches, since adding or changing components is as simple as adding or swapping modules. As will be shown in the power analysis section below, the modularity also decreases the total power consumption of the system for certain applications. Furthermore, modularity leads to better robustness and survivability since it eliminates single point of failure issues, to which centralized approaches are inherently susceptible. Finally, since the system is flexible and can adapt to many different possible applications, developing via the modular approach could give a lower long term programmatic cost. Several partially modular but inflexible architectures include the Sandia HERD nodes, the Berkeley Wireless Research Center picoRadio test beds, the Berkeley MICA motes, and the Rockwell WINS and Infocube platforms. The architecture proposed in Figure 1 is a series of operationally disjoint modules connected by a central bus, and satisfies many more of the desired system attributes than previous systems.

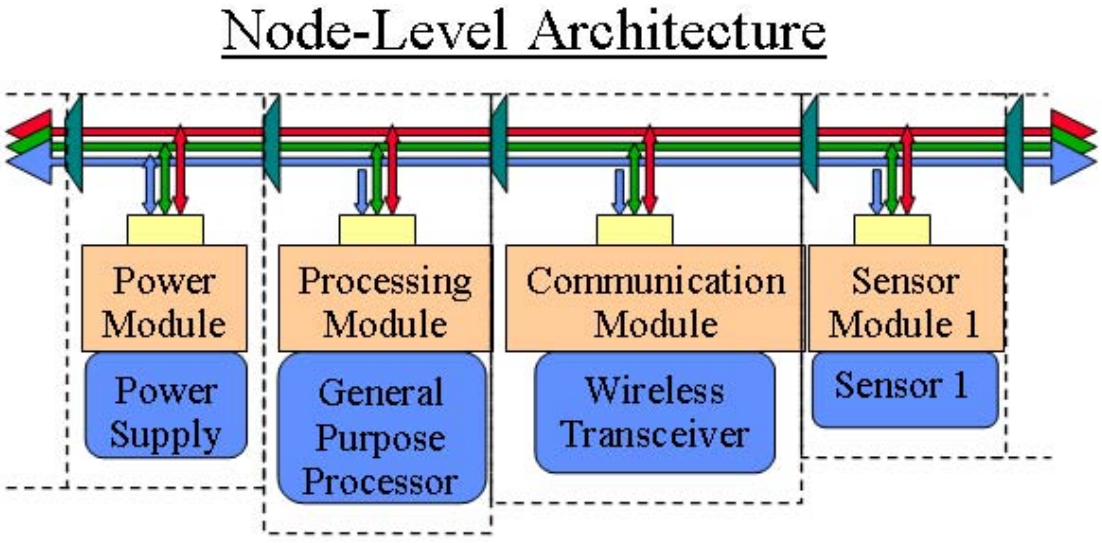

**Figure 1: Node System Architecture**

Each of the modules attached to the central bus acts as a stand alone component. The sensor modules each have their own data pre-processors (either low power general purpose processors, DSPs, micro-controllers, or FPGAs depending on level of computational complexity required) and data storage (either internal or external to the data pre-processor depending on memory requirements) as shown in Figure 2. This allows a higher power general purpose processor module to remain in a low power sleep mode for the majority of node operation. A general purpose processor is kept in the system in order to allow complex algorithms and node operations to occur when necessary thus providing a flexibility improvement over the application specific systems previously implemented. In order to reduce the processing requirements of the general purpose processing module further, the wireless networking module will handle all network routing when wireless messages simply need to be hopped on to other nodes.

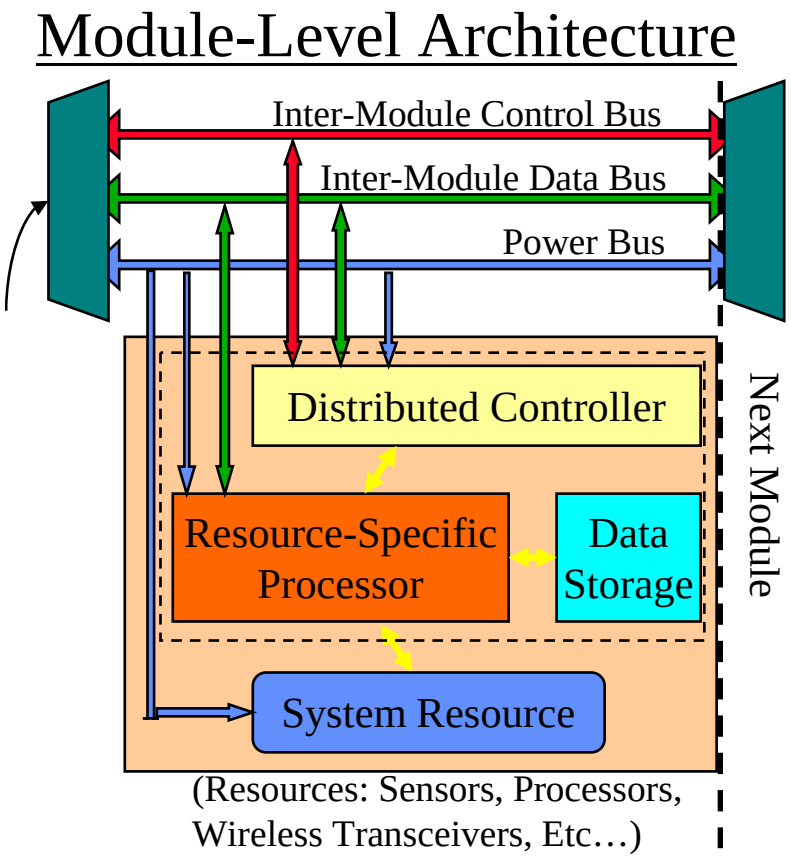


**Figure 2: Module Architecture**

The purpose of the sensor module data pre-processor will be two-fold: it will acquire and process the raw sensor data into a standard format, and it will act as an event detector. The advance event detection service that the pre-processor performs will take a first pass look at the data from the sensor to determine if an event has occurred. This first pass will likely take the form of threshold monitoring, envelope detection, or something similarly undemanding. When an event is registered, the sensor module will send a request to the general purpose processor module for verification. The processor module will gather the buffered data from the sensor module, fuse it with other relevant sensor data, and analyze it using higher-level computationally intensive algorithms to verify the event. If the event is verified, the general purpose processor could then pass along this high level situational information to other wireless nodes.

In order to make this architecture viable, each module must also have a separate hardware or software intra-node (inter-module) networking section, labeled as the Intra-Node Network Connector (INNC) in Figure 2. The INNC will be able to power on or off the module back-end and act as a gateway to the intra-node network. The INNC also protects the module from the heterogeneity of the back-ends of other modules, and allows for the extensibility and reconfigurability of the system. The INNCs are the only sections of the system which are required to be continuously powered and ready to receive interrupts. This requirement necessitates they be very low power and have the capability to enter interrupt ready sleep modes to conserve energy.

In the initial system, there will be four different groups of modules:

- Sensor modules (S)
- General purpose processor (application) modules (GPP)
- Wireless network communications modules (WNC)
- Power supply modules (PS)

Each of these modules will have an INNC for communication to all other modules. (Note: For the first prototype of MASS, there was only one bus to which all of the modules were connected. In future evolutions, multiple data bus channels will be included. Also, the PS does not have a bus interface or communications capabilities in the first prototype, but this possibility remains open for future developments as well.)

Nodes will consist of combinations of the different module types into various system configurations. There are three main classes of system configurations:

- Degenerate (D)
- Minimal (M)
- Complex (C)

Degenerate systems cannot support wireless sensor networks at all, minimal systems can contribute to wireless sensor networks only minimally, and complex systems can contribute to a wireless sensor network in some complex manner. Each of these classes breaks down into subclasses based on the particular modules that make up a system.

The three types of degenerate configurations are

- $D_1$: PS
- $D_2$: PS+S+(S,…)
- $D_3$: PS+GPP+(GPP,…)
- $D_4$: PS+S+GPP+(S,GPP,…)

The reason these systems are degenerate is that a fundamental premise of wireless sensor networks are communications between the nodes, at least in the sense of routing information, and these configurations have no WNC. The three types of minimal system configurations are:

- $M_1$: PS+WNC
- $M_2$: PS+WNC+S
- $M_3$: PS+WNC+GPP

A complex system will be defined as any system consisting of no GPP's and at least two S's, at most one GPP and at least one S, or at least two GPP's. Thus there will be three basic types of complex system configurations:

- $C_1$: PS+WNC+$S_1$+$S_2$+(S,…)
- $C_2$: PS+WNC+GPP+S+(S,…)
- $C_3$: PS+WNC+$GPP_1$+$GPP_2$+(S,GPP,…)

The initial phase of system operation will be a discovery of which type of system exists on a particular node. The system will support hot swapping of modules (i.e. changing system modules without cycling power or resetting the system), and graceful degredation (i.e. if a module fails during the course of system operation, this will not

bring down the entire system for certain failure modes). The operation of a module will change slightly based on what type of system it is in. This function changing will be explained in more detail in a section below.

# MATHEMATICAL SYSTEM ANALYSIS

## *POWER ANALYSIS*

### Module States:

In order to capture a general power model of each module, let each module have the following states:

| |
|---|
| ➢ transacting a request to or from another module,<br>➢ transacting a response to or from another module,<br>➢ processing requests,<br>➢ waiting for the bus to be available when wanting to transact a request or response,<br>➢ waiting for the destination module to be available when wanting to transact a request or response,<br>➢ and operating in a non-interacting state(s). |

**Table 1: Module States**

The full system will also consume power in the bus(es) that connect the modules. (If the power supply is centralized, which is likely based on simplicity and cost of implementation, there will also be additional power consumed due to the inefficiency of the power supply.) During request and response transacting, modules will be actively sending or receiving requests and responses from other modules via a system bus. A module may have to go into a wait state if either a bus or other module is unavailable for request or response transacting. During a wait state, a module will return to the standard non-interacting state from which it attempted to make a request or response. During the request processing state a module will perform any data manipulation or collection necessary in order to satisfy a request from another module. Finally, the module actions during a non-interacting state will depend on the specific module type. Data collection modules will be actively collecting, processing, and storing data or be in a deep sleep mode in the case of triggered sensors. Event detection modules will be collecting and processing data and monitoring for events. Processing modules will be in a deep sleep mode or performing some sort of periodic function. Communication modules will be routing network traffic and watching for messages from other nodes. A module may have multiple non-interacting states which it transitions between without any external involvement. Assume that the transition from state to state occurs in a very short period

of time so that it need not be considered in the system power consumption. Finally, let each module be able to request service from any other module except itself.

## Module Parameters and System Assumptions:

Based on this abstraction, the table below lists the fundamental decentralized system parameters.

| | |
|---|---|
| $p_{ij}^{t}$ (watts) | average power consumed by module i while transacting a request or response to or from module j (this may be dependent on module j because there may be several system buses each with different drivers) |
| $p_{i}^{w}$ (watts) | average power consumed by module i while in a wait state |
| $p_{i^k}^{n}$ (watts) | average power consumed by module i while in its $k^{th}$ non-interacting state |
| $p_{ij^k}^{p}$ (watts) | average power consumed by module i while processing a request from module j which was operating in its $k^{th}$ non-interacting state before the transaction |
| $d_{i^k j}^{q}$ (bits) | average amount of data sent from module i to module j during a request from i where i was operating in its $k^{th}$ non-interacting state before the transaction |
| $d_{ij^k}^{s}$ (bits) | average amount of data sent from module i to module j during a response from i where j was operating in its $k^{th}$ non-interacting state before the transaction |
| $r_{i^k j}$ (hertz) | average rate at which module i requests service from module j when module i operates in its $k^{th}$ non-interacting state |
| $w_{i}^{b}$ (seconds) | average wait time experienced by module i before being able to access a bus for transaction |
| $w_{ij}^{t}$ (seconds) | average wait time experienced by module i when attempting to transact with module j |

| | |
|---|---|
| $t_{ij^k}$ (seconds) | average time it takes module i to processes a request from module j when j was in its $k^{th}$ non-interacting state before transaction |
| $p_{ij}^{B}$ (watts) | average power consumed by a bus while it is being used to transact a message from module i to module j (this will likely be dependent on baud rate) |
| $B_{ij}$ (bits per second) | baud rate of bus used to transact a message from module i to module j |
| m (modules) | number of modules |

**Table 2: Decentralized System Parameters**

These parameters are meant to be assigned to constant values (or derived from constant values) in order to describe individual modules in the system. As such, this parameter set makes certain assumptions about the operation of the system. First, the system operation is assumed to be constant over time. For given system inputs, the system will give deterministic outputs, or at least outputs that have a deterministic average. While the times, powers, bits, and rates may randomly deviate slightly from these averages, they can be assumed for practical purposes to be constant. The only value that may deviate significantly from its average is $r_{i^k j}$, and this will be examined below.

Another major assumption is that the power consumption of the system is modal. In other words, when a module is in a certain mode of operation or in a certain operating state, the power a module consumes has at least a constant average. In other words, modules are considered as holistic entities rather than consisting of components that can operate independently. (Another approach would be to consider power consumption to be incremental. The power of a module here would be summed up over different independently operating components. In effect, the analysis that will be given could be pushed down to the scale of an a module with independent components adding another level to the module-node-network hierarchy.)

Another assumption of these parameters is that when operating in a given mode, each module interacts with other modules in only a single way or several ways that can be reasonably averaged together. In other words, if module i is in its $k^{th}$ mode and module j is in its $l^{th}$ mode, the amount of data passed between i and j during requests and

responses has a defined average. If there are several different types of requests module i might make of module j in a single mode, the average amount of data can be calculated as a weighted average of the data in each type of request where the weights derive from the particular rates of each type of request. Similarly, the overall request rate can be calculated as the sum of rates of request of each type of request. If reasonable, multiple transaction type modes could be broken into individual transaction type modes for analysis.

Furthermore, the processing and wait times must fall into some sort of distribution with a well defined average. In other words, all times in the system have at least a deterministic average. Additionally, any system buses are assumed to be serial. Parallel buses could be handled by mathematically breaking each parallel stream into its own bus, or by averaging the parallel bus into equivalent serial bus parameters. Finally, each request will have a non-periodic corresponding response of at least a request acknowledgment. A module may not request periodic responses from another module without any further requests. An initial request acknowledgement may be succeeded by a full request response at a later time. All parts of a request response are lumped into single parameters.

Other parameters are certainly more fundamental than these with regards to specific types of modules, but as for a general module parameter set, these variables should be sufficient. For example, for event detection modules, $r_{i^k j}$ can be derived from sensor sample rate, event rate, and probabilities of detection and false alarm. This type-specific breakdown of the parameters will be discussed in a subsequent section in more detail.

## System Power Consumption Derivation:

In order to develop the total power consumed by the system, the power will first be considered on a module by module basis in each of these states and then summed over the full system. In order to develop the average power consumed by a module, the power of each module state given in Table 1 must be analyzed. To capture the power consumption of the bus(es), it will be counted as a subset to a module's consumption

when that module is the initiator of the transaction. (This assumes that each message has one and only one source module. Bus power for broadcast messages are counted once and only once with this initiator module association.)

*Transacting a request to or from another module:*

Define an interval of length T. In this time period, requests will be made of module j at rates $r_{i^k j}$. The average number of requests from module i to module j during time period T will thus be $Tr_{i^k j}$. Similarly, the average number of requests from module j to module i will be $Tr_{j^k i}$. The time that each transaction takes will be determined by the amount of data sent and the baud rate of the bus. Specifically, for requests made of j, the time of each request will be $\frac{d^q_{i^k j}}{B_{ij}}$. For requests made by j, the time of each request will be $\frac{d^q_{j^k i}}{B_{ji}}$. Finally, the power consumed by module j while receiving or making a request will be $p^t_{ji}$. Combining these terms, the total power over an interval of length T consumed by module j while requesting services of or being requested by module i will be:

$$Tp^t_{ji}\left(r_{i^k j}\frac{d^q_{i^k j}}{B_{ij}}+r_{j^k i}\frac{d^q_{j^k i}}{B_{ji}}\right)$$

**Equation 1**

Including the power consumed by the bus for any requests made by module j, the total becomes:

$$T\left(p^t_{ji}\left(r_{i^k j}\frac{d^q_{i^k j}}{B_{ij}}+r_{j^k i}\frac{d^q_{j^k i}}{B_{ji}}\right)+r_{j^k i}p^B_{ji}\frac{d^q_{j^k i}}{B_{ji}}\right)=T\left(r_{i^k j}\frac{d^q_{i^k j}}{B_{ij}}p^t_{ji}+r_{j^k i}\frac{d^q_{j^k i}}{B_{ji}}\left(p^t_{ji}+p^B_{ji}\right)\right)$$

**Equation 2**

The average power is derived from taking this total power per module, dividing by the length of the interval, T, and summing over all modules, thus the average power consumed by module j while sending or receiving requests to or from other modules will be:

$$P_j^q = \sum_{i=1, i \neq j}^{m} \left( r_{i^k j} \frac{d_{i^k j}^q}{B_{ij}} p_{ji}^t + r_{j^k i} \frac{d_{j^k i}^q}{B_{ji}} \left( p_{ji}^t + p_{ji}^B \right) \right)$$

**Equation 3**

All of the following power derivations will follow a similar pattern. The power consumed, time the power is consumed, and rate at which module j is put into a certain consumption mode will be multiplied together and summed. Only if specific details are necessary will any detailed derivation be entered into.

*Transacting a response to or from another module:*

The rate at which module j will respond to a request by module i will simply be the rate at which module i requests module j. This observation is based on the assumption that every request begets a single response. Likewise, the rate at which module i will respond to module j will be the rate at which module j requests module i. The power consumed by the bus by responses from module j is calculated as in Equation 2. Overall, the power consumed by module j to send or receive responses to or from other modules, along with the associated bus power will be:

$$P_j^s = \sum_{i=1, i \neq j}^{m} \left( r_{i^k j} \frac{d_{ji^k}^s}{B_{ji}} \left( p_{ji}^t + p_{ji}^B \right) + r_{j^k i} \frac{d_{ij^k}^s}{B_{ij}} p_{ji}^t \right)$$

**Equation 4**

*Processing requests:*

The power consumed by processing requests is very straightforward. Combining the power module j requires to process a request from module i, the time it takes to process the request, and the rate of request, the total power module j consumes in processing requests is found to be:

$$P_j^p = \sum_{i=1, i \neq j}^{m} r_{i^k j} t_{ji^k} p_{ji^k}^p$$

**Equation 5**

*Waiting for the bus to be available when wanting to transact a request or response:*

The power consumed while waiting for a bus between j and i to be available is also a straightforward calculation. There are two parts to the power consumption, one for transacting a request and one for transacting a response. Combining these two results, the total power module j consumes while waiting for a bus to be available is found to be:

$$P_j^b = \sum_{i=1, i\neq j}^{m} \left( r_{j^k i} w_j^b p_j^w + r_{i^k j} w_j^b p_j^w \right) = \sum_{i=1, i\neq j}^{m} \left( w_j^b p_j^w \left( r_{j^k i} + r_{i^k j} \right) \right)$$

**Equation 6**

*Waiting for the destination module to be available when wanting to transact a request or response:*

The power consumed while waiting for a module to be available is also a straightforward calculation. There are two parts to the power consumption, one for transacting a request and one for transacting a response. Combining these two results, the total power module j consumes while waiting for other modules to be available is found to be:

$$P_j^t = \sum_{i=1, i\neq j}^{m} \left( r_{j^k i} w_{ji}^t p_j^w + r_{i^k j} w_{ji}^t p_j^w \right) = \sum_{i=1, i\neq j}^{m} \left( w_{ji}^t p_j^w \left( r_{j^k i} + r_{i^k j} \right) \right)$$

**Equation 7**

(In the system that will be implemented, the modules will not have a separate waiting state power since during the wait for a request to be processes, the module may carry out other functions. Specifically, the module may be in a non-interacting state, and thus for Equation 7 only, $p_j^w = p_{j^k}^n$, and the time $r_{j^k i} w_{ji}^t + r_{i^k j} w_{ji}^t$ can be subtracted from the summation term in Equation 8.)

*Operating in a non-interacting state(s):*

Finally, there will be periods during which module j will be operating independently of all other modules. This will only be during times when it is not transacting requests, transacting responses, processing requests, waiting for a bus to be available, or waiting for a destination module to be available. The cumulative amount of time during a period T during which the module will not be occupied will thus be:

$$T-\sum_{i=1,i\neq j}^{m}\left(r_{i^k j}\frac{d^q_{i^k j}}{B_{ij}}T+r_{j^k i}\frac{d^q_{j^k i}}{B_{ji}}T+r_{i^k j}\frac{d^s_{ji^k}}{B_{ji}}T+r_{j^k i}\frac{d^s_{ij^k}}{B_{ij}}T+r_{i^k j}t_{ji^k}T+r_{j^k i}w^b_j T+r_{i^k j}w^b_j T+r_{j^k i}w^t_{ji}T+r_{i^k j}w^t_{ji}T\right)$$

**Equation 8**

Simplifying Equation 8, this time becomes:

$$T-T\sum_{i=1,i\neq j}^{m}\left(r_{i^k j}\left(\frac{d^q_{i^k j}}{B_{ij}}+\frac{d^s_{ji^k}}{B_{ji}}+t_{ji^k}+w^b_j+w^t_{ji}\right)+r_{j^k i}\left(\frac{d^q_{j^k i}}{B_{ji}}+\frac{d^s_{ij^k}}{B_{ij}}+w^b_j+w^t_{ji}\right)\right)$$

**Equation 9**

During this time, the module will be consuming its non-interacting power. Multiplying by this power and dividing by the interval length, T, in order to find the average, the average power consumed by module j in a non-interacting state will be:

$$P^n_j=\left(1-\sum_{i=1,i\neq j}^{m}\left(r_{i^k j}\left(\frac{d^q_{i^k j}}{B_{ij}}+\frac{d^s_{ji^k}}{B_{ji}}+t_{ji^k}+w^b_j+w^t_{ji}\right)+r_{j^k i}\left(\frac{d^q_{j^k i}}{B_{ji}}+\frac{d^s_{ij^k}}{B_{ij}}+w^b_j+w^t_{ji}\right)\right)\right)p^n_{j^k}$$

**Equation 10**

$P^n$ will never be negative since it is not possible for a module to be busy for more than time T during a period of length T.

*Total System Power:*

Combining Equation 3-Equation 10, the total average power of the modular system, $P_m$, will be:

$$P_m=P^q_j+P^s_j+P^p_j+P^b_j+P^t_j+P^n_j$$

**Equation 11**

Which expands to:

$$P_m=\sum_{j=1}^{m}\left(p^n_{j^k}+\sum_{i=1,i\neq j}^{m}\left(r_{i^k j}\left(\left(\frac{d^q_{i^k j}}{B_{ij}}+\frac{d^s_{ji^k}}{B_{ji}}\right)\left(p^t_{ji}-p^n_{j^k}\right)+\left(w^b_j+w^t_{ji}\right)\left(p^w_j-p^n_{j^k}\right)+t_{ji^k}\left(p^p_{ji^k}-p^n_{j^k}\right)+\frac{d^s_{ji^k}}{B_{ji}}p^B_{ji}\right)\right.\right.$$
$$\left.\left.+r_{j^k i}\left(\left(\frac{d^q_{j^k i}}{B_{ji}}+\frac{d^s_{ij^k}}{B_{ij}}\right)\left(p^t_{ji}-p^n_{j^k}\right)+\left(w^b_j+w^t_{ji}\right)\left(p^w_j-p^n_{j^k}\right)+\frac{d^q_{j^k i}}{B_{ji}}p^B_{ji}\right)\right)\right)$$

**Equation 12**

As can be seen from Equation 12, if the power consumption of the modules never fluctuates from its non-interacting power consumption, i.e. if $p_{ij}^{t} = p_{i}^{w} = p_{i^k}^{n} = p_{ij^k}^{p}$, the total average power of the system will simply be:

$$P_m\Big|_{p_{ij}^{t}=p_i^{w}=p_{i^k}^{n}=p_{ij^k}^{p}} = \sum_{j=1}^{m} p_{i^k}^{n} + \sum_{j=1}^{m}\sum_{i=1,i\neq j}^{m}\left( p_{ji}^{B}\left( r_{i^k j}\frac{d_{ji^k}^{s}}{B_{ji}} + r_{j^k i}\frac{d_{j^k i}^{q}}{B_{ji}} \right)\right)$$

**Equation 13**

This power is simply the sum of the power consumption of each module plus the power consumed by transactions on the bus(es). This configuration is similar to the case of a centralized system in which no components ever fluctuate in power.

## Centralized System Power:

In order to compare a standard centralized architecture with the decentralized one presented, a similar mathematical development must be made for the centralized system. This discussion is identical to that given in a previous published paper by the authors. Assume that all components of the centralized system are attached to a single central processor that has two modes, a processing mode and a sleep mode. Whenever the processor is not servicing a component or analyzing data collected from components, it is in sleep mode, otherwise it is in its processing mode. Assume the transitions between the two modes are close to instantaneous. Based on this abstraction, Table 3 illustrates the six centralized system parameters .

| | |
|---|---|
| ${}^{c}t_i$ | time required to service component i and analyze resulting data |
| ${}^{c}r_i$ | rate at which component i must be serviced |
| ${}^{c}p_p$ | central processor active power consumption |
| ${}^{c}s_p$ | central processor sleep power consumption |

| | |
|---|---|
| ${}^{c}p_{ci}$ | component i power consumption |
| n | number of components |

**Table 3: Centralized System Parameters**

First, the total processing power, ${}^{c}p$, and standby power, ${}^{c}s$, of the centralized system must be calculated:

$$ {}^{c}p = {}^{c}p_{c} + \sum_{i=1}^{n} {}^{c}p_{c_i} \quad and \quad {}^{c}s = {}^{c}s_{c} + \sum_{i=1}^{n} {}^{c}p_{c_i} $$

**Equation 14**

Following a similar derivation to those above, the average power that the centralized architecture will spend in its processing mode will be:

$$ \sum_{i=1}^{n} r_i t_i \, {}^{c}p $$

**Equation 15**

and the average power spent in its standby mode will be:

$$ \left(1 - \sum_{i=1}^{n} r_i t_t\right) {}^{c}s $$

**Equation 16**

Combining these two equations, the total power for a centralized system will be:

$$ P_c = {}^{c}s + \sum_{i=1}^{n} {}^{c}r_i \, {}^{c}t_i \left({}^{c}p - {}^{c}s\right) $$

**Equation 17**

In order to generate meaningful comparisons, simulations of the centralized and decentralized power models were performed. Parameter values used for all of the simulations were kept constant unless the parameter was used as the independent variable. The parameter values used are shown in Table 4.

| **Centralized** | **(All components are identical)** |
|---|---|
| Central Processor Standby Power | 100uW |
| Central Processor Processing Power | 250mW |
| Number of External Components | 4 |

| | |
|---|---|
| Component Sampling Rate | 5Hz |
| Sample Servicing Time | 10ms |
| Component Power | 5mW |
| **Decentralized** | **(All non-GPPM modules are identical)** |
| General Purpose Processor Module (GPPM) Standby Power | 100uW + 1mW = 1.1mW |
| GPPM Processing Power | 250mW + 15mW = 265mW |
| Number of Modules (Including GPPM) | 4 + 1 = 5 |
| Rate Each Module Requests GPPM Validation (i.e. Event Rate) | .4Hz |
| Validation Request Service Time | 100ms |
| Non-GPPM Module Standby Power | 5mW + 1mW = 6mW |
| Non-GPPM Module Processing Power | 5mW + 15mW = 20mW |

**Table 4: Simulation parameters**

Resulting plots are shown in Figure 3 below. The general result is that there are certain applications for which a centralized system would be lower and power and others for which the decentralized system would be lower power. The decentralized system is lower power for applications requiring high sensor sampling rates, low environmental event rates (i.e. low rates of requiring GPP validation), low sensor power, and high sensor service times. The centralized system is lower power for applications requiring low sensor sampling rates, high environmental event rates, high sensor power, and low sensor service times. In general, the breakdown seems to clearly place centralized systems into data collection applications and decentralized systems into in-network event detection applications.

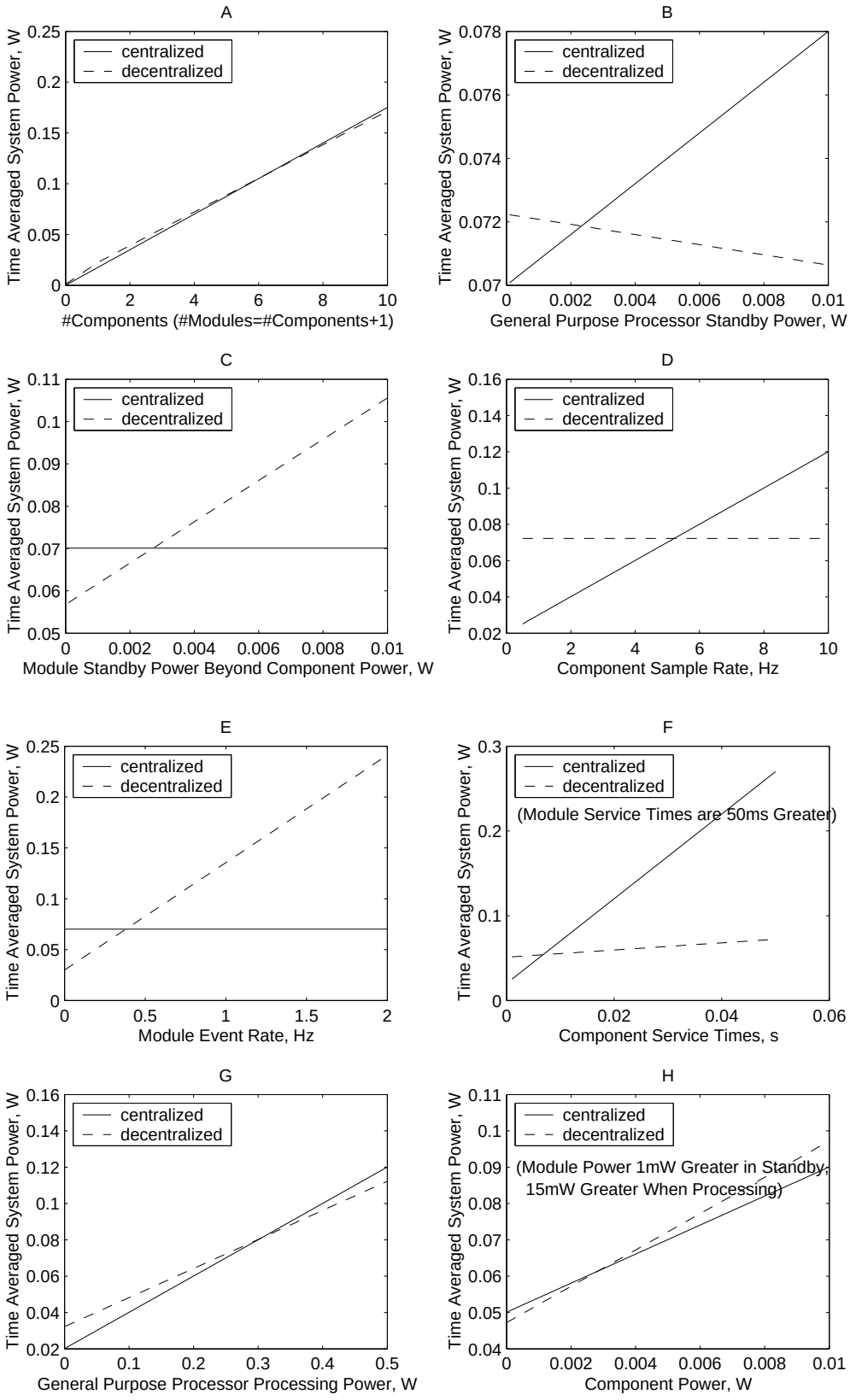


**Figure 3: Comparison of centralized and decentralized system power**

## *Minimum Power System Configuration:*

The purpose of a mathematical model for system power in the decentralized system is to use it to find techniques to minimize the power consumption of the system. Tradeoffs embedded in Equation 12 indicate that there may be an optimal mode for each module in the system, and a system configuration so that the system consumes a minimum amount of power. This section analyzes tradeoffs and control methods for the modular architecture.

### Processing Time vs. Processing Power:

The first tradeoff to be analyzed is that between processing time and processing power. A well known relationship between power consumption of digital circuits and frequency of digital circuits is that power is directly proportional to frequency. Assuming a given process takes a given time at a certain frequency, power will be inversely related to the length of time a process requires to execute. Assuming a processor is mounted on a module that has a power consumption consisting of some minimum static power, S milliWatts, plus a power that is linearly dependent on frequency with slope F mW/MHz, the execution time for a given process and module power can be related through:

$$t_{ji^k} = \frac{F}{p^{p}_{ji^k} - S}$$

**Equation 18**

F may be determined experimentally if it is not given on the data sheets of the module's components. (Current standard low end values of F are around .5mW/MHz for microprocessors.)

### Wait Time for the Bus:

There is also a relationship between wait times for the bus(es) and the bus usage for requests and responses. As evident in the derivation of total system power above, the

total amount of time the bus will be used in a period of length T by a module other than module j will be:

$$T \sum_{l=1,l\neq j}^{m} \left( r_{l^k j} \frac{d^q_{l^k j}}{B_{lj}} + r_{j^k l} \frac{d^s_{lj^k}}{B_{lj}} + \sum_{i=1,i\neq\{j,l\}}^{m} \left( r_{i^k l} \left( \frac{d^s_{li^k}}{B_{li}} + \frac{d^q_{i^k l}}{B_{il}} \right) + r_{l^k i} \left( \frac{d^s_{il^k}}{B_{il}} + \frac{d^q_{l^k i}}{B_{li}} \right) \right) \right)$$

**Equation 19**

(This assumes a single system bus.) Thus, by the Monte Carlo method, the probability that the bus is busy when module j attempts to access it any given time will be:

$$q^b_j = \sum_{l=1,l\neq j}^{m} \left( r_{l^k j} \frac{d^q_{l^k j}}{B_{lj}} + r_{j^k l} \frac{d^s_{lj^k}}{B_{lj}} + \sum_{i=1,i\neq\{j,l\}}^{m} \left( r_{i^k l} \left( \frac{d^s_{li^k}}{B_{li}} + \frac{d^q_{i^k l}}{B_{il}} \right) + r_{l^k i} \left( \frac{d^s_{il^k}}{B_{il}} + \frac{d^q_{l^k i}}{B_{li}} \right) \right) \right)$$

**Equation 20**

And correspondingly, the probability that the bus is free is:

$$p^b_j = 1 - q^b_j = 1 - \sum_{l=1,l\neq j}^{m} \left( r_{l^k j} \frac{d^q_{l^k j}}{B_{lj}} + r_{j^k l} \frac{d^s_{lj^k}}{B_{lj}} + \sum_{i=1,i\neq\{j,l\}}^{m} \left( r_{i^k l} \left( \frac{d^s_{li^k}}{B_{li}} + \frac{d^q_{i^k l}}{B_{il}} \right) + r_{l^k i} \left( \frac{d^s_{il^k}}{B_{il}} + \frac{d^q_{l^k i}}{B_{li}} \right) \right) \right)$$

**Equation 21**

A standard wired MAC (Medium Access Control) for a decentralized controlled system is CSMA (Carrier Sense Multiple Access). The modules will sense the bus to see if it is being used, and if it is not, immediately start using it. There are additions, such as Ready-To-Send (RTS) and Clear-To-Send (CTS) messages that can be used to ensure collision-less communication as well as collision detection (CD) and acknowledge / non-acknowledge (ACK/NACK) routines to recover from collision if it does occur. If the bus is busy when it is first sensed, a common method of retrying is for module j to wait an exponentially distributed random amount of time, such as:

$$t^b_j = \frac{1}{\lambda^b_{j^k}} e^{-t/\lambda^b_j}$$

**Equation 22**

where $\lambda^b_{j^k}$ is both the average and standard deviation of the retry times. (Note: These retry time parameters, $\lambda^b_{j^k}$, may be different for each module, or each mode of each module, and may also differ based on the priority of the request.) Assuming that communications on the bus will be fast, and that the retry time average, $\lambda^b_{j^k}$, will be long

compared to the length of the bus communications (so that access attempts are essentially independent trials), the expected wait time for the bus will be:

$$w_j^b = \lambda_{j^k}^b \frac{q_j^b}{p_j^b}$$

**Equation 23**

Shorter retry periods will not reduce the wait time to zero as Equation 23 would indicate. As the retry periods get shorter, the assumption of the independence of the access attempts breaks down since two or more attempts may be made during the same busy period. (The full form of the wait time is quite complex and will not be derived here.) This causes the minimum expected wait time as $\lambda_{j^k}^b$ goes to zero to be:

$$w_{j\min}^b = \frac{1}{2q_j^b} \sum_{l=1,l\neq j}^{m} \sum_{i=1,i\neq\{j,l\}}^{m} \left( r_{i^k l} \left( \left( \frac{d_{li^k}^s}{B_{li}} \right)^2 + \left( \frac{d_{i^k l}^q}{B_{il}} \right)^2 \right) + r_{l^k i} \left( \left( \frac{d_{il^k}^s}{B_{il}} \right)^2 + \left( \frac{d_{l^k i}^q}{B_{li}} \right)^2 \right) \right)$$

**Equation 24**

Which is simply the average length of time to the end of a busy period if the bus is found to be busy on a first attempt.

## Wait Time for a Module:

Another relationship can be determined between average wait times for other modules in the system and processing times and request rates. This wait time is intimately related to the method by which modules request each other. As explained above, each module will have a queue into which other modules may insert themselves with a given priority of service. The queue is setup so that no module gets completely ignored in the queue, even if the priority of its request is low. The problem of the wait time for a given module thus becomes a fairly complex queuing theory question.

The primary variables in a queuing theory question are the inter-arrival times and the service times of the queue. Inter-arrival time is the time between requests made on the module. Service time is the amount of time it takes a module to process a request. These are both considered random variables with a distribution that must be determined. A

reasonable assumption for distribution of inter-arrival times is an exponential distribution with an aggregate arrival rate of:

$$\lambda^{t}_{Total,j} = \sum_{l=1, l\neq j}^{m} \lambda^{t}_{l^k} = \sum_{l=1, l\neq j}^{m} r_{l^k j}$$

**Equation 25**

In other words a distribution of:

$$M\left(x;\lambda\right) = \lambda^{t}_{Total,j} e^{-x\lambda^{t}_{Total,j}}$$

**Equation 26**

This distribution essentially assumes that requests are mostly closely spaced in time and rarely are they separated by a long period of no requests. It also corresponds to assuming a Poisson distribution for the arrival times into the system. These assumptions seem reasonable because it is likely that if a request just occurred because there was an event in the environment, that event will likely cause another request in a short time. In periods of no environmental activity, there will be no requests or only very infrequent spurious requests made, but with no regularity.

As for service times, a reasonable assumption is that the service times for a given type of request will be normally distributed. The service time may not be exactly constant due to slight variations in the speed of the processor, scaling of the processor speed, or other variables. The standard deviation of the service times will likely be very small, however, so a sharp Gaussian distribution is expected. In order to utilize queuing theory, the distribution of all of the service times must be found. The most common general way to find the aggregate service time distribution is to simply average the distributions of all of the service times together. The distribution of the average of several Gaussians will itself be Gaussian. (In fact, due to the Central Limit Theorem, the average of several variables, regardless of their own underlying distributions, will be Gaussian.) Queuing theory doesn't generally deal with Gaussian distributions, but a similarly shaped distribution that it does deal with is the Erlangian distribution. Trying to minimize the integral of the squared difference between an Erlangian distribution and a Gaussian distribution in general is an algebraically intractable problem. Only guidelines for the Erlangian distribution parameters can thus be given, but for the specific set of queuing that is likely to result in the modular architecture proposed, these guidelines should be

sufficient. If the average of the Erlangian distribution is set equal to the average of the Gaussian distribution, one way to match the distributions is to match the height of their peaks. Doing this and solving for the Erlangian shaping factor, r, gives:

$$\frac{\sqrt{2\pi}r\left(r-1\right)^{r-1}e^{-(r-1)}}{\left(r-1\right)!}=\frac{\mu}{\sigma}$$

**Equation 27**

The limiting behavior of this complex expression for r is

$$\sqrt{r}=\frac{\mu}{\sigma}$$

**Equation 28**

Which is better than 5% accurate already for r=5, and also sets the standard deviation of the two curves equal to each other. Thus the approximation between Gaussian and Erlangian distribution is:

$$E\left(x;r,\mu\right)=\frac{\frac{\mu}{\sigma^2}\left(\frac{\mu}{\sigma^2}x\right)^{\frac{\mu^2}{\sigma^2}-1}}{\left(\frac{\mu^2}{\sigma^2}-1\right)!}e^{-\frac{\mu}{\sigma^2}x}\approx\frac{1}{\sigma\sqrt{2\pi}}e^{-\left(\frac{x-\mu}{2\sigma}\right)^2}=G\left(x;\sigma,\mu\right),\quad x\geq 0$$

**Equation 29**

The average, $\mu$, of this distribution, which will differ for each module, can be simply calculated as the weighted average of service times:

$$\mu_j=\frac{\sum_{l=1,l\neq j}^{m}r_{l^k j}t_{jl^k}^{Total}}{\sum_{l=1,l\neq j}^{m}r_{l^k j}}$$

**Equation 30**

Where

$$t_{jl^k}^{Total}=t_{jl^k}+\frac{d_{l^k j}^{q}}{B_{lj}}$$

**Equation 31**

Is the total service time including the transferring of the service request on the bus. The standard deviation of the requests, $\sigma_j$, cannot be explicitly calculated, but a reasonable assumption might be that it is on the order of 5% of $u_j$.

From introductory queuing theory for the M/E/1 queue, the total service time (wait time in queue plus processing time), $w_{ij}^t$, and average queue length, $L_j$, will thus be:

$$L_j = \rho + \rho^2 \frac{1 + \frac{\sigma_j^2}{\mu_j^2}}{2(1-\rho)}$$

**Equation 32**

And

$$w_{ij}^t = \mu_j + \frac{\rho\mu_j\left(1 + \frac{\sigma_j^2}{\mu_j^2}\right)}{2(1-\rho)}$$

**Equation 33**

Where

$$\rho_j = \lambda_j^t \mu_j = \sum_{l=1, l\neq j}^{m} r_{l^k j} t_{jl^k}^{Total}$$

**Equation 34**

Is the service load on module j and must be less than 1. The load factor, $\rho_j$, essentially represents the amount of service time (in seconds) requested of module j per second. $\lambda_j^t$ is the average total arrival rate of requests on module j and is given by:

$$\lambda_j^t = \sum_{l=1, l\neq j}^{m} r_{l^k j}$$

**Equation 35**

Since the queue in module j will be finite with some length $K_j$, there may be periods in which service is completely refused and a module i must retry its request. In general, the probability that the queue is full, i.e. $P\left(L_j \geq K_j\right)$, is:

$$P\left(L_j \geq K_j\right) = 1 - \sum_{k=0}^{K_j - 1} Z^{-1}\left( \frac{\left(1-\rho_j\right)\left(1-z\right)\left(\frac{r}{r+\rho_j-\rho_j z}\right)^r}{\left(\frac{r}{r+\rho_j-\rho_j z}\right)^r - z}, k \right)$$

**Equation 36**

Where $Z^{-1}$ is the inverse z-transform. This is in general not a computable answer, but an approximation can be made. If an assumption of $\sigma_j = 5\%\mu_j$ is made, then $r = 400$. As $r \to \infty$, the Erlangian distribution approaches an impulse at $\mu_j$ which is simply a deterministic service time system. In the case for r large, Equation 36 simplifies to:

$$P\left(L_j \geq K_j\right) = 1 - \left(1-\rho_j\right)\sum_{k=0}^{K_j - 1} \rho_j^k = \rho_j^{K_j}$$

**Equation 37**

Thus the probability that module i is turned away from module j is simply:

$$P_j^{Busy} = \rho_j^{K_j} = \left( \sum_{l=1, l\neq j}^{m} r_{l^k j} t_{jl^k}^{Total} \right)^{K_j}$$

**Equation 38**

The waiting time thus needs to be revised to take this into account. If module i retries module j at an average rate of $\lambda_{i^k j}^R$ (which may come from an exponential distribution), the total service time can be revised as:

$$w_{ij}^t = \lambda_{i^k j}^R \frac{\rho_j^{K_j}}{1-\rho_j^{K_j}} + \frac{\rho_j \mu_j \left(1 + \frac{\sigma_j^2}{\mu_j^2}\right)}{2\left(1-\rho_j\right)} + \mu_j$$

**Equation 39**

Thus far a non-prioritized, first-come-first-served queue has been assumed. The queuing discipline described above, however, does have prioritization. With prioritization, the average waiting time will change to yet another form. The greatest difficulty with calculating the wait time for the specific prioritized scheme proposed is that prioritization with finite storage means that there is the possibility of a service request getting bumped off the queue once that request is already on it. This probability is complex to calculate primarily because the distribution of the number of requests in the

queue when a given request arrives plus the number of requests that arrive above a given request after it arrives is non-trivial. If a deterministic average service time is assumed (due to an extremely sharp Erlangian service time distribution), a concrete result appears, however. The average slot a priority p request will enter into the queue will be:

$$N_p = \frac{{}_p\rho}{1 - {}_p\rho}$$

**Equation 40**

Where

$${}_p\rho = \mu \sum_{c=p}^{P} {}_c\lambda^t = \sum_{l=1, l\neq j}^{m} {}^c r_{l^k j} \, {}^c t_{jl^k}^{Total}$$

**Equation 41**

And ${}_c\lambda^t = \sum_{l=1, l\neq j}^{m} {}^c r_{l^k j}$ denotes the average arrival rate of requests with priority c. (Equation 41 implicitly assumes that the average processing time of all requests is the same regardless of priority of the request.) If there are K total spots in the queue, then the average number of spots left in the queue after an arrival will be $K - \lfloor N_p + 1 \rfloor$. Also, the average length of time remaining for a request already being processed will simply be $\mu/2$. Thus the probability of a request with priority p being bumped will be:

$$P_p^{Bumped} = P^{K-\lfloor N+1 \rfloor+1}\left(\frac{\mu}{2}\right) + P^{K-\lfloor N+1 \rfloor+2}\left(\frac{3\mu}{2}\right) + \ldots + P^{K}\left(\mu\left(\frac{1}{2} + \lfloor N+1 \rfloor - 1\right)\right)$$

**Equation 42**

Where

$$P^n(t) = 1 - e^{-{}_p\lambda^t t} \sum_{i=0}^{n-1} \frac{\left({}_p\lambda^t t\right)^i}{i!}$$

**Equation 43**

Is the probability of at least n arrivals from a Poisson arrival process in an interval (0,a). Putting Equation 42 and Equation 43 together results in:

$$P_p^{Bumped} = \sum_{j=0}^{\lfloor N_p+1 \rfloor - 1} \left( 1 - e^{-{}_p\lambda^t \mu\left(\frac{1}{2}+j\right)} \sum_{i=0}^{K-j} \frac{\left( {}_p\lambda^t \mu\left(\frac{1}{2}+j\right)\right)^i}{i!} \right)$$

**Equation 44**

From introductory queuing theory, the average waiting time in a prioritized queuing system with no storage bound is computed recursively from:

$$W^p = \frac{\sum_{c=1}^{P} {}_c\lambda^t \left( {}_c\sigma^2 + {}_c\mu^2 \right)}{\left( 1 - \sum_{c=p}^{P} {}_p\rho \right)\left( 1 - \sum_{c=p+1}^{P} {}_p\rho \right)}$$

**Equation 45**

Where P is the maximum priority, ${}_c\sigma^2$ is the variation of processing times for priority c requests (also equal to ${}_c\mu^2 / r$ for the Erlangian distribution with parameter r), and ${}_l\mu$ is the mean of processing times for priority c requests. Assuming that the variation and mean of each type of request is similar, Equation 45 can be simplified to:

$$W^p = \frac{\left(\sigma^2 + \mu^2\right)\sum_{c=1}^{P} {}_c\lambda^t}{\left( 1 - \mu\sum_{c=p}^{P} {}_c\lambda^t \right)\left( 1 - \mu \sum_{c=p+1}^{P} {}_c\lambda^t \right)}$$

**Equation 46**

The total average wait time for a given module will thus be:

$${}_p w_{ij}^t = \frac{\lambda_{i^k j}^R \frac{\rho_j^{K_j}}{1-\rho_j^{K_j}} + W_j^p}{1 - P_p^{Bumped}} + \mu_j$$

**Equation 47**

This wait time expression is obviously quite complicated. This level of analysis is probably unnecessary for the modular system described, but it does allow a way for the variable $w_{ij}^t$ to be removed from the expression for the power of the modular system. Simulations will be performed to verify these results, and in general, the simulations will be used as the benchmark for how the actual operation will occur.

One additional constraint that could be put on determining this wait time is that the population from which requests occur might be considered finite. Since modules may make more than one request of other modules, however, the number of requests that could come in is, strictly speaking, infinite. The wait time expression given in Equation 47 is thus the correct wait time for the system. For sake of completeness, the wait time for an M/M/1/K queue will be cited. This queue has exponential distributions of inter-arrival times and service times, a single request processor, and a total of K modules that may request another module only once each. It is not prioritized. The wait time expression is:

$$ {}_{p}w_{ij}^{t} = \frac{L}{\lambda_{Total,j}^{t}\left(m-L\right)} + \mu_{j} $$

**Equation 48**

Where

$$ L = \frac{\sum_{l=1,l\neq j}^{m} l^{2}\left(l-1\right)!\binom{m}{l}\left(\frac{\lambda_{Total,j}^{t}}{\mu_{j}}\right)^{l}}{\sum_{l=1,l\neq j}^{m} l!\binom{m}{l}\left(\frac{\lambda_{Total,j}^{t}}{\mu_{j}}\right)^{l}} $$

**Equation 49**

In order to show the benefits of the queuing approach as described in Equation 47 over a simpler queue-less approach, another brief analysis must be performed. Assume that there are no queues in the system at all so that if a module is busy processing another request, a requesting module must simply retry later. The first thing to note is that priority scheduling cannot be enforced in this type of queue-less system, and this is an immediate initial drawback. The probability that a node is busy in this system is simply its traffic load:

$$ P_{j}^{Busy} = \rho_{j} $$

**Equation 50**

Assume a module retries with an average rate of $\lambda_{i^{k}j}^{R} = c\mu_{j}$ where c is a positive constant and $\mu_{j}$ is the average processing time on module j. The wait time for this system can then be found to be:

$$w_{ij}^{t} = \begin{cases} \dfrac{\rho_j}{1-\rho_j}\left(c\mu_j\right)+\mu_j, & for \quad c \geq 1 \\ \dfrac{\rho_j}{1-\rho_j}\left(\left\lceil \dfrac{1}{c} \right\rceil c\mu_j\left(1-c\left\lfloor \dfrac{1}{c} \right\rfloor\right)+\dfrac{c^2\mu_j\left\lfloor \dfrac{1}{c} \right\rfloor\left(\left\lfloor \dfrac{1}{c} \right\rfloor+1\right)}{2}\right)+\mu_j, & for \quad c<1 \end{cases}$$

**Equation 51**

Which for $c=\frac{1}{s}$ , where s is an integer when it is greater than 1, simplifies to:

$$w_{ij}^{t} = \begin{cases} \dfrac{\rho_j}{1-\rho_j}\left(\dfrac{\mu_j}{s}\right)+\mu_j, & for \quad s \leq 1 \\ \dfrac{\rho_j}{1-\rho_j}\dfrac{\mu_j\left(s+1\right)}{2s}+\mu_j, & for \quad s>1 \end{cases}$$

**Equation 52**

An interesting result of this is that as $s \to \infty, \quad c \to 0, \quad \lambda_{i^k j}^{R} \to 0$ , i.e. as the retry period becomes very short, is:

$$\lim_{\lambda_{i^k j}^{R} \to 0} w_{ij}^{t} = \frac{\rho_j}{1-\rho_j}\frac{\mu_j}{2}+\mu_j$$

**Equation 53**

This is the shortest possible expected wait time in the queue-less system. This result makes sense because $\frac{\mu_j}{2}$ is the average length of time until the end of a processing interval if module j was busy when another module originally attempted access.

Comparing Equation 52 with Equation 47, the wait time difference between the two module interaction policies (prioritized queuing or non-queuing) can be determined. The question is whether:

$$\frac{\lambda_{i^k j}^{R}\dfrac{\rho_j^{K_j}}{1-\rho_j^{K_j}}+W_j^{p}}{1-P_p^{Bumped}} < \frac{\rho_j}{1-\rho_j}\lambda_{i^k j}^{R}$$

**Equation 54**

The several factors affecting this inequality are the average processing time of requests, the deviation in average processing time of requests, the number of priorities used in a

prioritization scheme, the retry interval for requests, the distribution of traffic across priorities, and the size of the queue in the queuing scheme. The figures below demonstrate how each one of these factors independently affects which of the two schemes is better. The graphs show the ratio of non-queuing total wait time to prioritized queuing total wait time across the different priorities, i.e.:

$$\frac{\lambda^{R}_{i^k j}\dfrac{\rho_j^{K_j}}{1-\rho_j^{K_j}}+W_j^{p}}{\dfrac{1-P_p^{Bumped}}{\dfrac{\rho_j}{1-\rho_j}\lambda^{R}_{i^k j}+\mu_j}}+\mu_j$$

When the graph is above 1, the prioritized queuing scheme is thus a better choice, and these portions are colored yellow. When the graph is below 1, the non-queuing scheme is a better choice, and these portions are colored blue. Unless the parameter was being varied, it was set to the middle of the range given when it was varied. For example, the processing time was set to .05, 50ms, for all simulations other than that generating Figure 4.

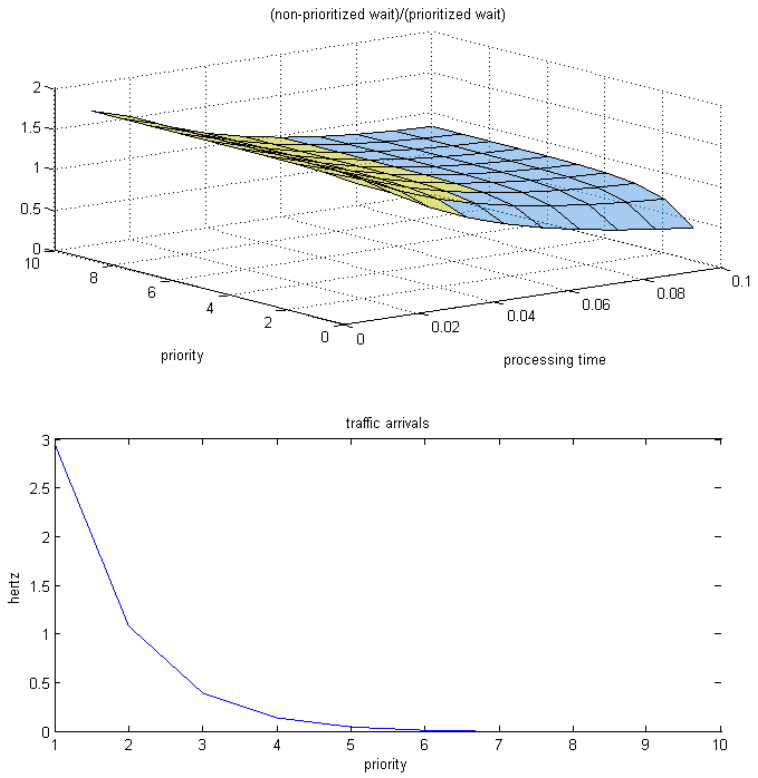


**Figure 4: Processing time versus priority for queuing and non-queuing schemes**

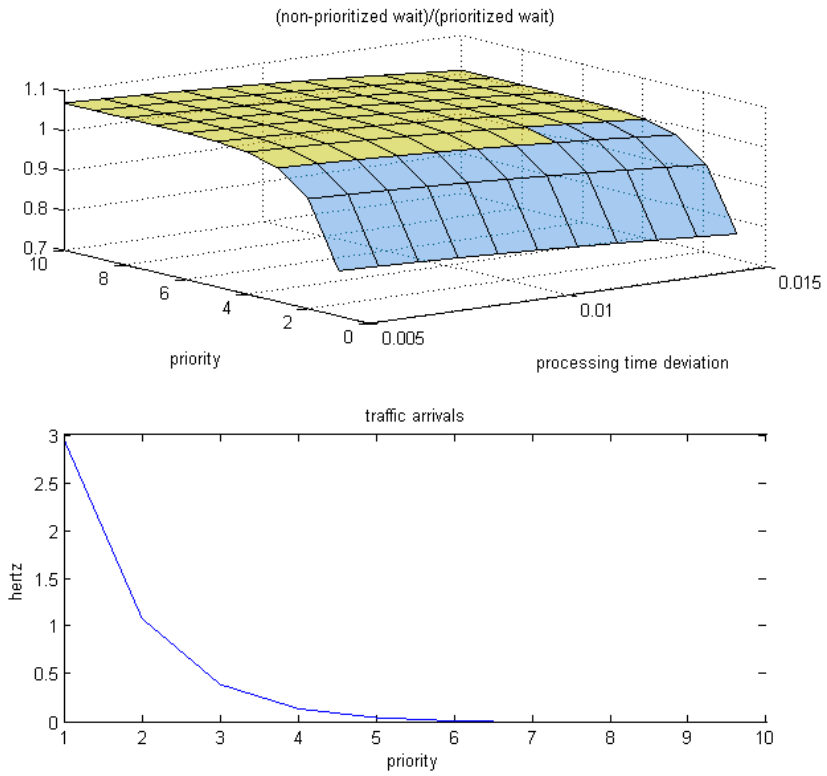


**Figure 5: Processing time deviation versus priority for queing and non-queuing schemes**

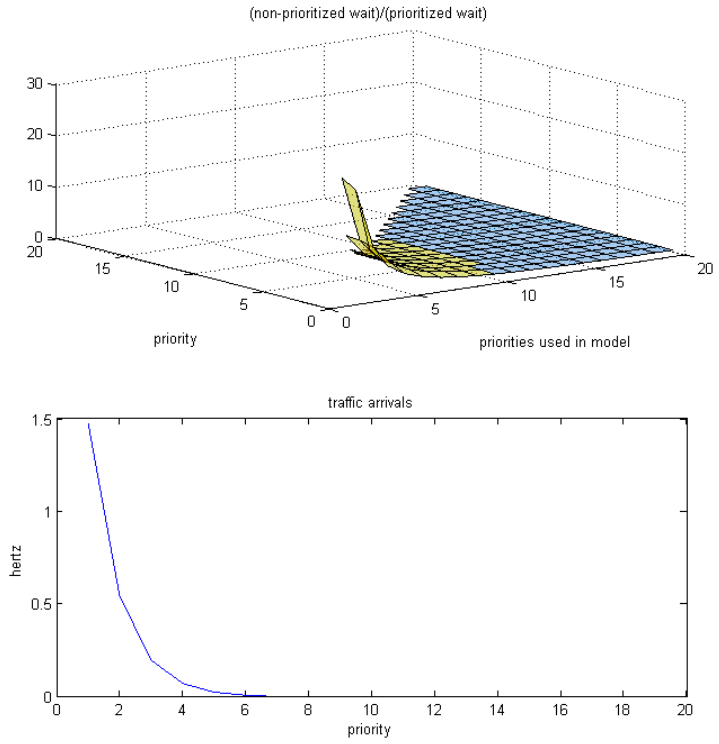


**Figure 6: Priorities used in prioritized scheme versus priority for queing and non-queuing schemes**

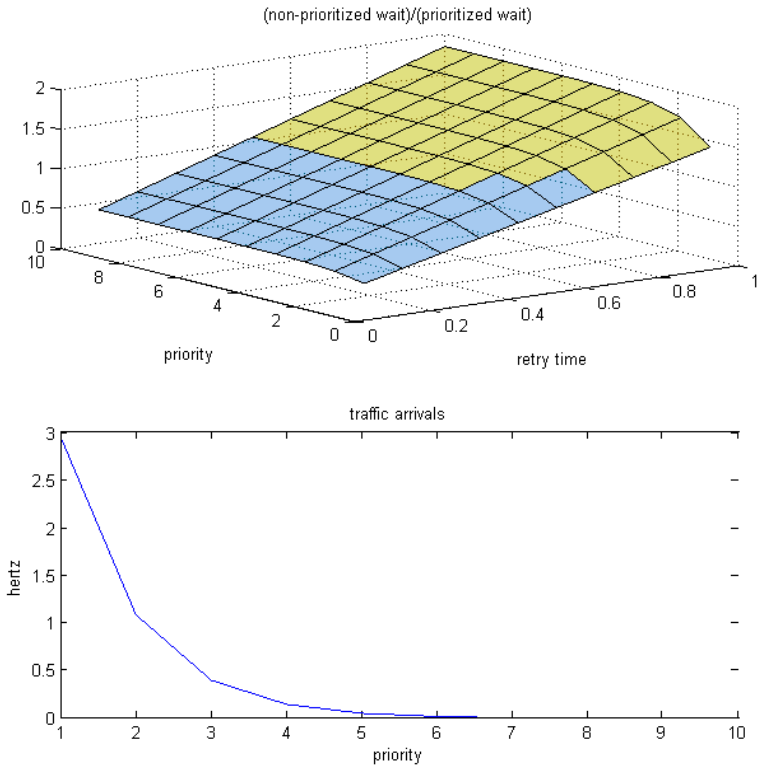


**Figure 7: Request retry time versus priority for queing and non-queuing schemes**

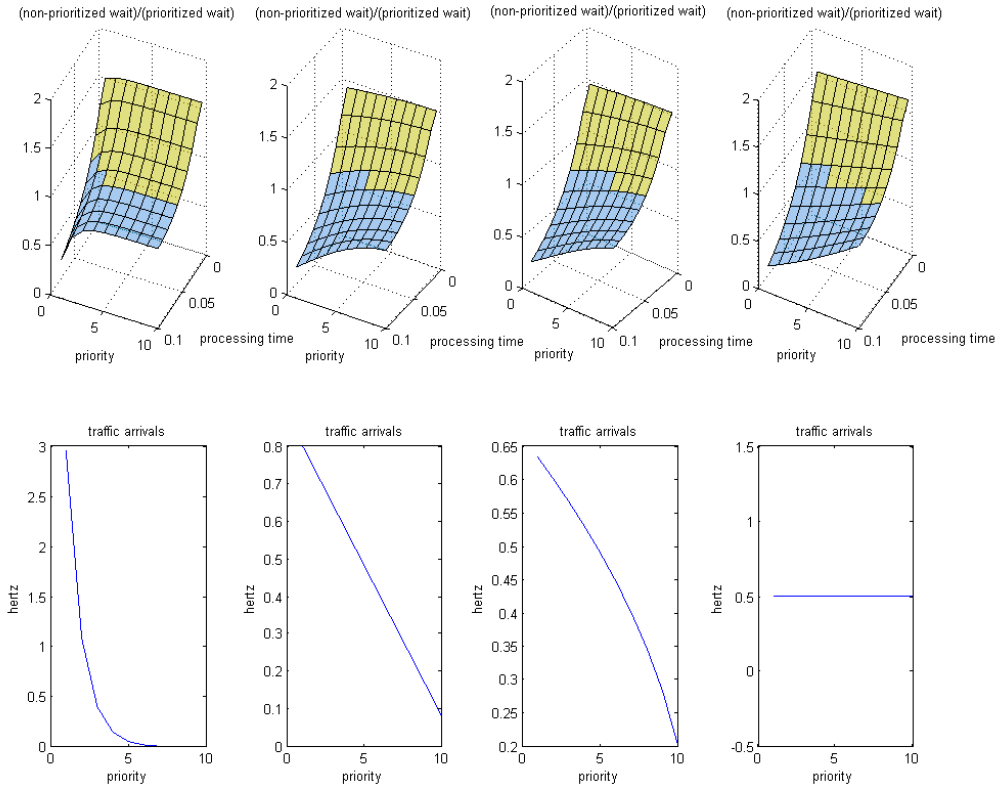


**Figure 8: Processing time versus priority for different traffic models**

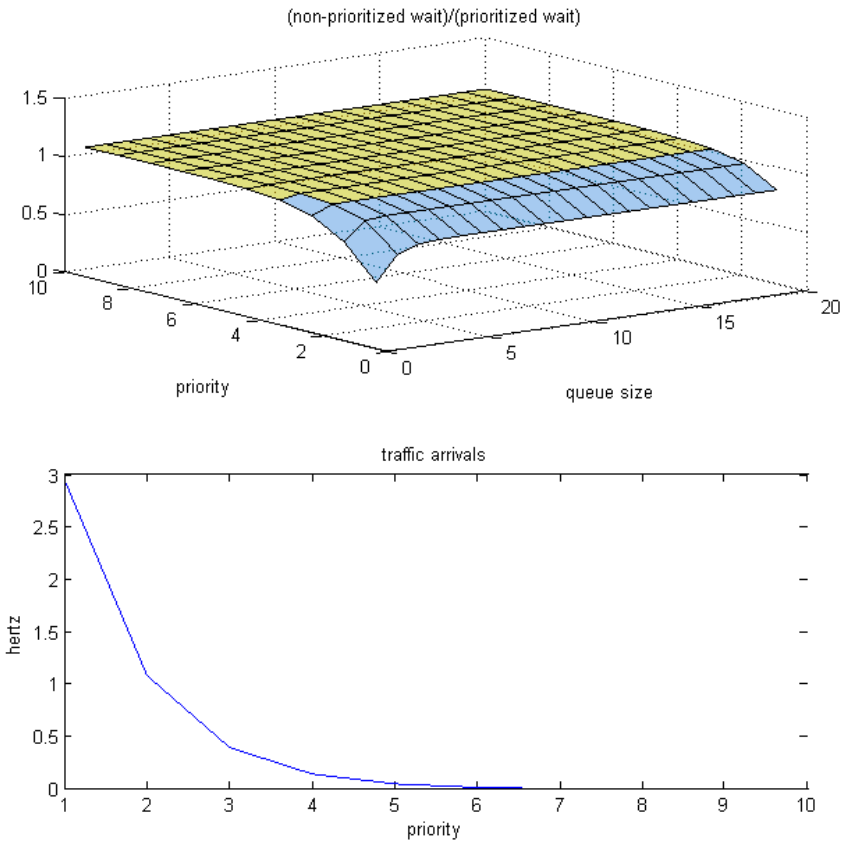


**Figure 9: Queue size versus priority for queing and non-queuing schemes**

These figures offer a wealth of information about queuing versus non-queuing schemes in general. They will be interpreted only at a topical level here, but much deeper understandings are likely to emerge on further analysis. From Figure 4 and Figure 5, it is apparent that for situations in which requests have a low average processing time, but significant deviation away from an average, the priority queuing scheme is more beneficial. For a high performance processing resource taking in various different kinds of data from multiple resources, this will likely be the situation. In fact, in high traffic situations, the lower priorities would favor a non-queuing scheme whereas the high priorities favor a queuing scheme. Figure 6 shows that the priority queuing scheme is essentially only beneficial if a low number of priority levels are used. In other words, priority queuing is a good choice only if a very low number of priority levels (e.g. 3) are used. Figure 7 shows that as retry time increases lower average wait times will be experienced by requests using the queuing scheme, and this result is fairly consistent

across priority level. Figure 8 shows that as traffic distribution across priority levels becomes imbalanced towards lower priority requests, the priority queuing system becomes a more favorable scheme. In other words, if the number of high priority requests is significantly smaller than the number of low priority requests, and high priorities are used sparingly, the queuing scheme becomes more favorable. Finally, Figure 9 shows that the size of the queue in a queuing scheme generally has no major effect on whether to choose a queuing or non-queuing scheme except for low priorities and low queue sizes. For low priorities and low queue sizes, the non-queuing scheme is favored. This is likely a result of low priority requests being bumped more often for low queue sizes. Other than Figure 4 and Figure 5 which offer support towards using the priority queuing scheme based on the expected usage of node resources, the other simulations simply provide guidelines as to how best to utilize the priority queuing scheme assuming it is in place. Thus for implementation purposes, the priority queuing system will be chosen.

One other concern that Equation 54 raises is that evidently as $\rho \to 1$ or $P_p^{Bumped} \to 1$, low priority requests will statistically never be serviced. This is only the case if a certain percentage of the requests are high priority and the queue size is small, however. Specifically, if ${}_p\lambda^t\mu = a\lambda^t\mu$, i.e. if higher priority traffic makes up 100a% of the total traffic, the queue size K must be greater than a threshold value, depending on the total traffic $\rho$, in order to probabilistically ensure some minimum $P_p^{Bumped}$. The simulation results shown below demonstrate how these minimum supporting queue sizes, K, vary with $a$, $\rho$, and $P_p^{Bumped}$.

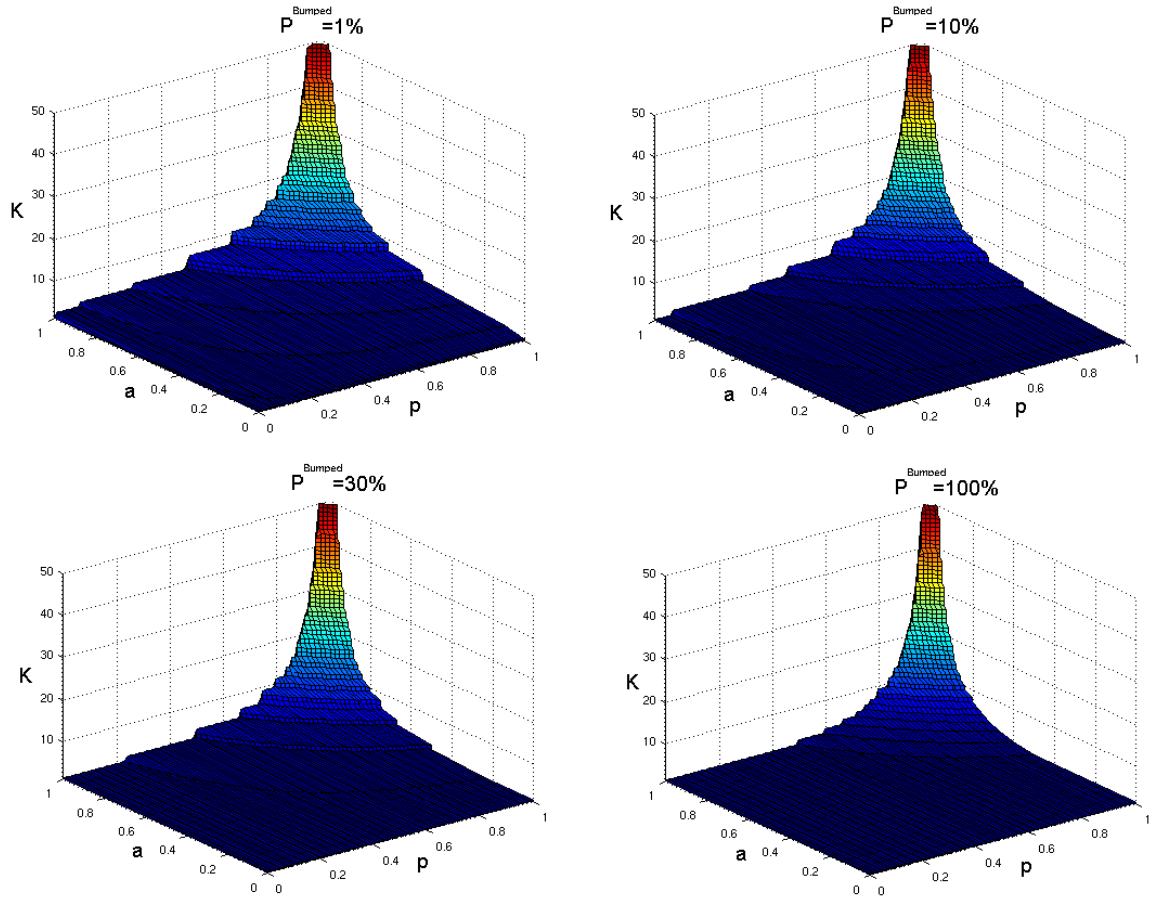


**Figure 10: Minimum Supporting Queue Sizes for Ensuring a Maximum Probability of Bumping**

As expected, as $P_p^{Bumped}$ is allowed to increase, the minimum required queue size for a given $a$, $\rho$ combination decreases. Also as expected, as either $a$ or $\rho$ increases while holding the other parameters constant, the required minimum queue size increases as well. The simulations reveal that the minimum required queue size increases very sharply as $a \to 1$ and $\rho \to 1$. They also reveal that $a$ and $\rho$ are essentially symmetric parameters. In other words, if specific values of $a$ and $\rho$ are interchanged with each other, the required queue size, K, does not change. In fact, Equation 44 demonstrates that if the product $a\rho$ is a constant, no matter what the explicit values of $a$ or $\rho$, the queue size required will be the same. The required queue size thus depends only on the amount of traffic that is at a higher priority than a specific request, not on the total traffic into the module. The required queue size such that requests of all priorities will have less than a specific $P_p^{Bumped}$ is simply the case when $a \to 1$ in Figure 10.

Figure 10 can help choose the appropriate queue size once a specific $a$, $\rho$, and $P_p^{Bumped}$ are specified. A reasonable queue size might be about 10 giving $a\rho = .75$ for $P_p^{Bumped} = 10\%$, but the specific choice might depend on empirical testing since many of the effects of queue size will be transition effects, but all of the equations are developed under steady-state operating assumptions. Additionally, affects that will occur in the specific implementation of the modular architecture may not be captured by the somewhat simplified queuing model developed. (Specifically, although the population from which requests are generated is not strictly finite, since modules may make multiple requests, it will likely depend in practice on the number of modules in the system and each of their bandwidths. A finite request population will serve to reduce the necessary queue size in general.)

Revisiting Equation 54, and making the assumptions that $P_p^{Bumped} < 10\%$ and $K > 10$, the equation can be simplified to an approximate requirement on $\rho$:

$$\rho > \frac{W_j^p}{\lambda_{i^k j}^R \left(1 - P_p^{Bumped}\right) + W_j^p}$$

**Equation 55**

In other words, if the traffic, $\rho$, is above a certain threshold, the prioritized queuing system will provide lower wait times than the non-queuing system. As $W_j^p$ gets smaller, i.e. as the average wait time in the queue (queuing time) for a priority p request gets smaller, Equation 55 dictates that a lower traffic, $\rho$, is necessary such that the prioritized queuing system generates lower waiting times than the non-queuing system. Contrastingly, as $W_j^p$ gets larger, Equation 55 dictates that a larger traffic, $\rho$, is necessary before the queuing system becomes a lower waiting time system than the non-queuing system. Also, as $\lambda_{i^k j}^R$ increases or $P_p^{Bumped}$ decreases, the threshold for $\rho$ decreases. In other words, for high priority traffic, the queuing system will provide lower wait times than the non-queuing system for most traffic, but for lower priority traffic, the queuing system will provide higher wait times than the non-queuing system unless the traffic is very high. Also as the retry-frequency increases or the probability of being bumped off the prioritized queue decreases, lower and lower levels of traffic will still

favor implementing the prioritized queuing system over the non-queuing system. There is thus a tradeoff between being able to ensure prioritized requests and having balanced wait times across all types of requests.

## Inter-module request rate, sample rate, and event rate:

The system as designed is meant to be completely event driven. Although the architecture can be adapted to other, data collection type, activities, the power analysis shows that lower power can be achieved with a centralized system for high event (sample) rates. (A data collection system could be morphed into an event detection system by considering an event to occur when incoming data deviated significantly from a previous measurement.) The inter-module request rates, $r_{i^k j}$, will depend on a few module-specific parameters: environmental event rate, $e_{ij}$, the sensing sample rate, $r_{i^k}^s$, the probability of detection, $p_{i^k j}^d$, and the probability of false alarm, $p_{i^k j}^f$. (These parameters assume that there is only one sensor per module, there can only be one sample rate of the sensor, but there can be multiple ways the collected data is processed in order to determine different types of events which will be reported to different modules.) The sample rate, probability of detection, and probability of false alarm are all dependent on the operational mode, k, of the module. The power consumed by a module to perform an event detection calculation is assumed to increase as the probability of detection increases, the probability of false alarm decreases, and the sensing sample rate increases.

A simple derivation is needed in order to determine the relationship between $r_{i^k j}$, $e_{ij}$, $r_{i^k}^s$, $p_{i^k j}^d$, and $p_{i^k j}^f$. Define an interval of length T. Let $M_k$ be the total number of samples in the interval, $E_{ij}$ be the total number of events in the interval which may cause module i to request service of module j, N be the total number of non-events, and $R_{i^k j}$ be the total number of requests generated to module j by module i. Evidently:

$$M_k = E_{ij} + N$$

**Equation 56**

Also, the number of requests, $R_{i^k j}$, can be calculated simply as:

$$R_{i^k j} = p^d_{i^k j} E_{ij} + p^f_{i^k j} N$$

**Equation 57**

Combining Equation 56 and Equation 57, dividing by the interval T, and rearranging gives:

$$r_{i^k j} = p^d_{i^k j} e_{ij} + p^f_{i^k j} \left( r^s_{i^k} - e_{ij} \right)$$

**Equation 58**

Since $r^s_{i^k} \geq e_{ij}$, the sample rate is greater than or equal to the detected event rate, the request rate will always be positive. Equation 58 demonstrates that if the sensing sample rate increases, the request rate will increase also due to false alarm requests. $r_{i^k j}$, $r^s_{i^k}$, and $p^f_{i^k j}$ can all be measured by a module, where the probability of false alarm use module j's processing result as absolute determination of whether an event occurred or not. Given a receiver operating characteristic (ROC) for the relationship between $p^d_{i^k j}$ and $p^f_{i^k j}$, Equation 58 gives a direct method of calculating the true event rate, $e_{ij}$, into the system.

$$e_{ij} = \frac{r_{i^k j} - p^f_{i^k j} r^s_{i^k}}{p^d_{i^k j} - p^f_{i^k j}}$$

**Equation 59**

(In general $p^d_{i^k j} \geq p^f_{i^k j}$, if $p^f_{i^k j} = 0$ then $p^d_{i^k j} = 0$, and if $p^d_{i^k j} = 1$ then $p^f_{i^k j} = 1$. An example ROC is given in Figure 11 below. As the decision regions for events and non-events overlap more and more, i.e. as the signal to noise ratio for a particular sensed signal becomes large, the ROC becomes $p^d_{i^k j} = p^f_{i^k j}$. This situation implies that any signal entering the system can be defined as an event, and thus $e_{ij} \to \infty$ as shown by Equation 59. The verification of events by module j will still leave some total probability of false alarm for the system, but Equation 58 is the best guess the system can generate for $e_{ij}$.)

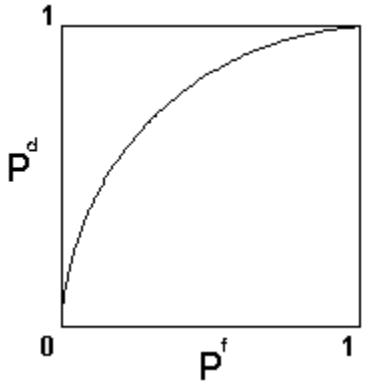


**Figure 11: Example Receiver Operating Characteristic**

In order to dynamically adjust the mode of the module, a relationship must be established as to how to change $r_{i^k}^s$ as $e_{ij}$ changes. In other words, what algorithm will be used to ensure that $r_{i^k}^s \geq be_{ij}$, the sample rate is some multiple of the event rate. One very simply algorithm is as follows:

$$r_{i^k}^s{}_{n+1} = \min\left(r_{i^k}^s{}_{\max}, \max\left(r_{i^k}^s{}_{\min}, b\left(e_{ij}{}_n + d\left(e_{ij}{}_n - e_{ij}{}_{n-1}\right)\right)\right)\right)$$

**Equation 60**

$e_{ij}{}_n$ is the estimated event rate and is equal to the number of events per sample from the last m samples taken, and minimum and maximum limits on the sample rate are applied. The relationship between d in Equation 60 and the number of samples, m, in the moving average $e_{ij}{}_n$ was determined experimentally with simulations in order to give the minimum cumulative error, see Figure 12 (d=5, b=2, $r_{i^k}^s{}_{\max} = 2.5$, $r_{i^k}^s{}_{\min} = .5$), Figure 13, and Figure 14. In general as d gets smaller, and thus the number of samples in the moving average gets smaller, the response of the system to changes in the environment gets faster. As d gets smaller, however, the system also gets more sensitive to spurious events and random changes in the event rate. The determination of the d to use should be made empirically depending on the type of events observed. b should also be determined per application since a high b may use undue amounts of processing power, but a low b risks missing events.

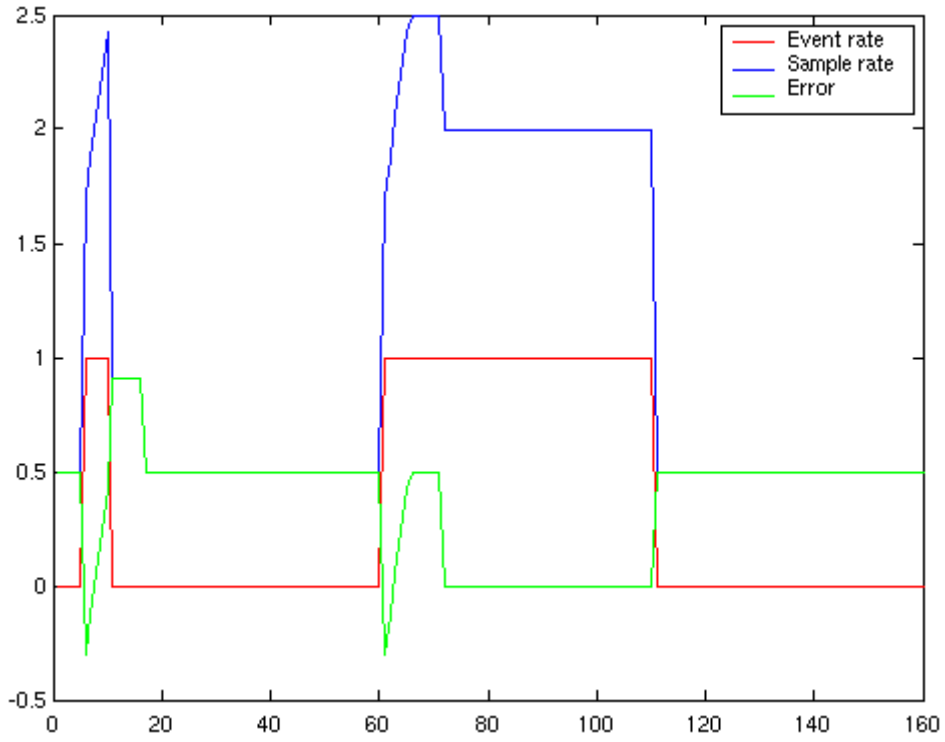


**Figure 12: Sample Rate Tracking the Event Rate**

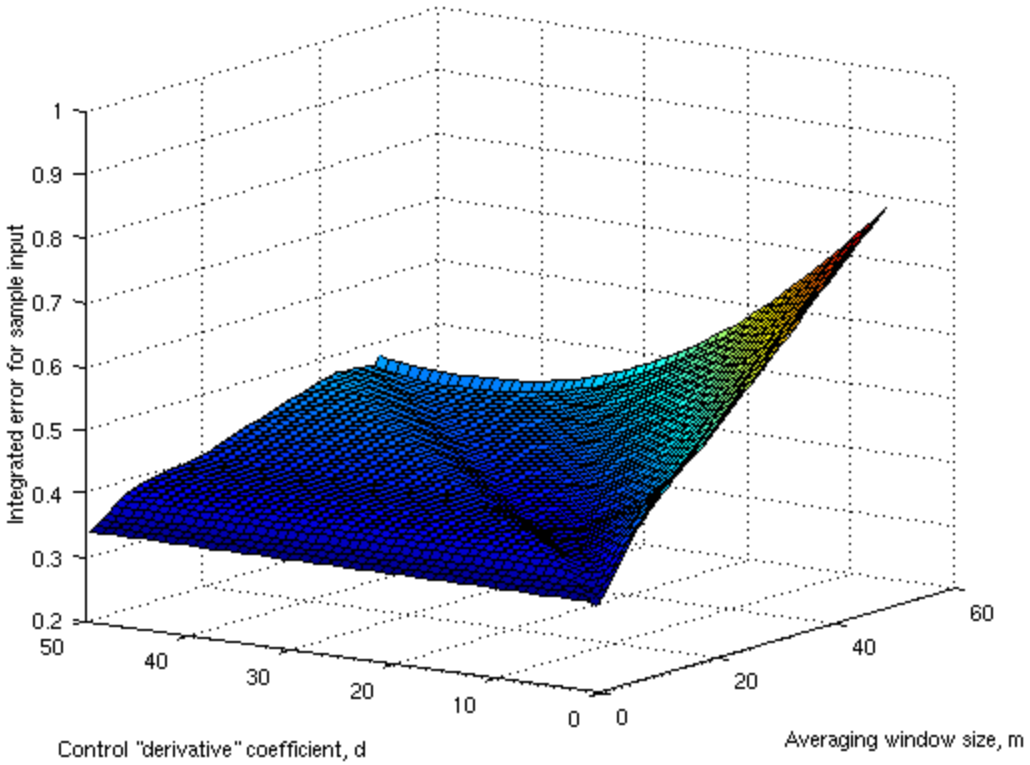


**Figure 13: Cummulative error versus averaging window size, m, and control "derivative" coefficient, d, for sample input**

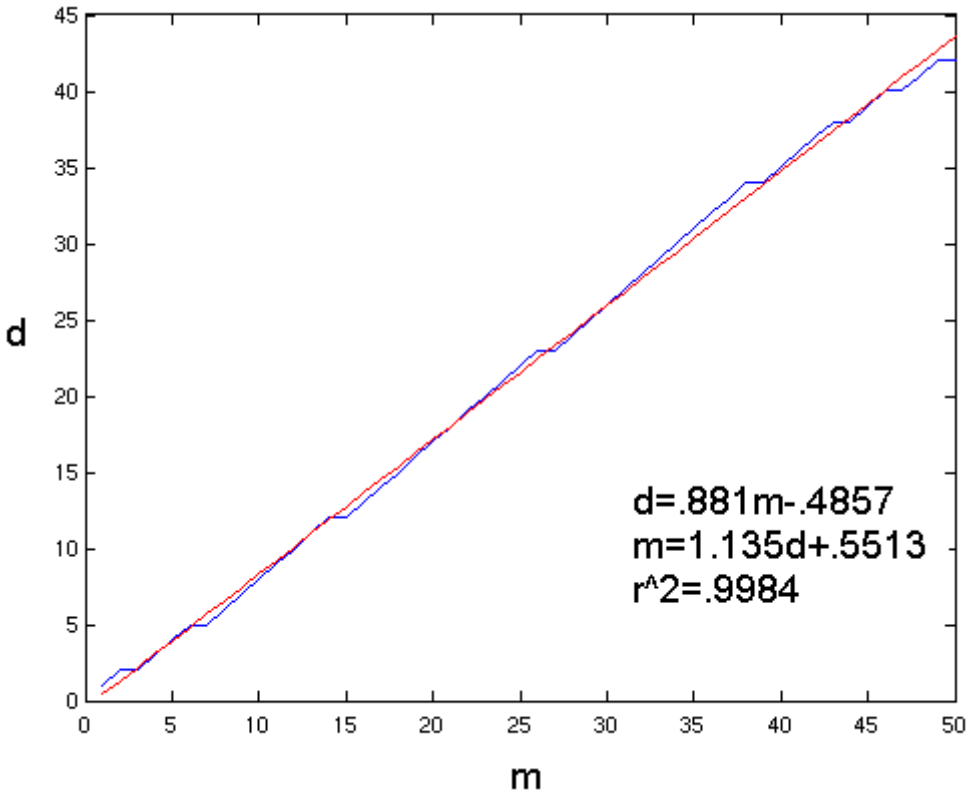


**Figure 14: Optimal control "derivative" coefficients, d, for varying averaging window size, m, with regression analysis**

Minimization of total power consumption:

The optimal control of the node has the singular goal of using the least amount of power necessary while still accomplishing its given mission. Since the node has no central controller, however, the optimal control must be pushed down to decisions on the individual modules. The individual modules seek to minimize the amount of power they cause the system to consume which includes their own power, the power used to transact their requests to other modules, and the power used by other modules to process their requests. This power can be given by:

$$P_i = p_{i^k}^n + \sum_{j \neq i}^{m} r_{i^k j} \left( t_{ji^k} p_{ji^k}^p + j_{i^k j}^{trans} \right)$$

**Equation 61**

where $j_{i^k j}^{trans}$ is the amount of power consumed in transacting the request both by the modules involved and by the media that transacts the request. For sensor modules operating on nodes with a general purpose processor module, $j_{i^k j}^{trans}$ will simply be:

$$j_{i^k j}^{trans} = \frac{d_{i^k j}^{q}}{B_{ij}} \left( p_{ij}^{t} + p_{ij}^{B} + p_{ji}^{t} \right)$$

**Equation 62**

If there is no local general purpose processor and the data must be shipped to another node for processing, however, $j_{i^k j}^{trans}$ is given by:

$$j_{i^k j}^{trans} = \frac{d_{i^k j}^{q}}{B_{ij}} \left( p_{ij}^{t} + p_{ij}^{B} + p_{ji}^{t} \right) + j^{wireless\_comms} \approx j^{wireless\_comms}$$

**Equation 63**

since wireless communication is generally consumes vastly more power than any onboard computation or bus communication. As given in [1], the communications power is given by:

$$j^{wireless\_comms} = \frac{L}{R} \left( P_{TX} + P_{OUT} \right) + P_{TX} T_{tx->st} + P_{RX} \left( \frac{L}{R} + T_{rx->st} \right)$$

**Equation 64**

for a single slot, single packet transmission. L is the length of a packet in bits, R is the baud rate of the radios, $P_{TX/RX}$ is the power consumed by the transmitter/receiver, $P_{OUT}$ is the output power of the transmitter, and $T_{tx/rx->st}$ is the start-up time of the transmitter/receiver.

All of the variables and sub-variables of Equation 61 have now been tied to fundamental measurable parameters of the system except for $p_{i^k}^{n}$. (The fundamental measurable parameters can either be measured off-line as in the case of $p_{ij}^{B}$, for example, or they may be controllable parameters of the system, such as $r_{i^k}^{s}$.) The two dynamically adjustable parameters on the sensor modules of the system are $r_{i^k}^{s}$ and $p_{i^k j}^{f}$, and while the power consumed by other modules is already related to these via $r_{i^k j}$ in Equation 58, $p_{i^k}^{n}$

has not yet been related to these parameters to form the controllable balance the optimal power can be derived from.

A definitive functional form to relate $p_{i^k}^n$ to $r_{i^k}^s$ and $p_{i^k j}^f$ is not completely forthcoming. A reasonable assumption is that there will be some amount of power consumed by a module independent of either $r_{i^k}^s$ or $p_{i^k j}^f$, and then other components of the power that will be. This provides a first pass relationship of

$$p_{i^k}^n = P_{i^k}^{Base} + f\left(r_{i^k}^s, p_{i^k j}^f\right)$$

**Equation 65**

where $P_{i^k}^{Base}$ is the measurable base power of the module independent of $r_{i^k}^s$ and $p_{i^k j}^f$. Another reasonable assumption is that the dependence of $p_{i^k}^n$ on $r_{i^k}^s$ will be linear. This is likely because if a fixed amount of processing is required per collected sample, and if there is a fixed average power per instruction, then as additional samples are processed in the same time period, the increase in processing instructions will be proportional and thus the increase in power will be proportional as well.

As for $p_{i^k j}^f$, a relationship is less obvious. Firstly, if the complexity of an algorithm increases, the number of instructions the algorithm requires to execute will also increase, but the link between complexity of an algorithm and the $p_{i^k j}^f$ it will provide is difficult to firmly establish. For example, how does one compare the complexity of a simple amplitude threshold in the time domain to a 16 point DFT in the frequency domain? The $p_{i^k j}^f$ in these two cases depend strongly on the type of noise in the surrounding environment and also the type of signal that is trying to be detected. Even attempting to quantify the general difference in $p_{i^k j}^f$ between a 16 point DFT and a 256 point DFT is difficult without knowing more about the specific problem and algorithm used to detect a target. In general, $p_{i^k j}^f$ is usually calculated using Monte Carlo methods on simulated inputs, and no analytic analyses are possible.

Since there is no analytic analysis possible, a reasonably upper-bound assumption for the dependence of $p_{i^k}^n$ on $p_{i^k j}^f$ will be given. Assume an algorithm with processing

time $O\left(n^2\right)$, where n corresponds to some sort of "resolution" of the algorithm. (For a 16 point DFT, the "resolution" would be 16 since the analysis is broken into 16 discrete frequencies. A radix-2 n-point DFT is actually better with a $O\left(n\lg\left(n\right)\right)$ in processing time.) Furthermore, assume that as the resolution is doubled, the probability of false alarm only halves. Generally, as resolution is doubled, the probability of false alarm decreases by well more than half, so this is simply a reasonable upper bound. Thus, $p^f_{i^k j} \propto \frac{1}{n}$. This gives an overall inverse quadratic relationship between $p^n_{i^k}$ and $p^f_{i^k j}$ in a worst case. Since the justification of the linear relationship between $p^n_{i^k}$ and $r^s_{i^k}$ involves the number of instructions executed per sample, and the relationship of $p^n_{i^k}$ and $p^f_{i^k j}$ is derived through this instructions per sample ratio, a reasonable assumption is that $r^s_{i^k}$ and $p^f_{i^k j}$ will be together in a single term in the function relating them to $p^n_{i^k}$. Thus

$$p^n_{i^k} = P^{Base}_{i^k} + r^s_{i^k} \sum_{j=1;\, j\neq i}^{m} \frac{c_{ij}}{\left(p^f_{i^k j}\right)^2}$$

**Equation 66**

where $c_{ij}$ is a constant of proportionality.

Putting together Equation 58, Equation 61, and Equation 66, the power a module must seek to minimize will be:

$$P_i = P^{Base}_{i^k} + r^s_{i^k} \sum_{j=1;\, j\neq i}^{m} \frac{c_{ij}}{\left(p^f_{i^k j}\right)^2} + \sum_{j=1,\, j\neq i}^{m} \left(p^d_{i^k j} e_{ij} + p^f_{i^k j}\left(r^s_{i^k} - e_{ij}\right)\right)\left(t_{ji^k} p^p_{ji^k} + j^{trans}_{i^k j}\right)$$

**Equation 67**

In order to place a lower bound on $r^s_{i^k}$, consider that the sample rate should be fast enough as to catch any event. In other words, if events have a minimum time duration, the sample rate should be statically minimum where it is at least the inverse of this duration time. Also, if a module detects an event every time it senses its environment, it may very well be missing events in between samples. Thus $r^s_{i^k}$ should be dynamically adjusted above the static minimum so that it samples its environment say at least some

constant, b, times as frequently as it detects events, i.e. $r_{i^k}^s \geq be_{ij}$. The maximum $r_{i^k}^s$ may go is simply equal to the maximum samples per second throughput of the algorithm used to process the samples. The analysis of how to change $r_{i^k}^s$ in relation to $e_{ij}$ has already been determined above. The restriction on $p_{i^k j}^f$ is simply $0 \leq p_{i^k j}^f \leq 1$.

From Equation 67, it is apparent that minimizing $r_{i^k}^s$ is always preferable since it only appears linearly. Thus $r_{i^k}^s$ is chosen in relation to the types of events to be detected, not in relation to power consumption considerations. Also, each $p_{i^k j}^f$ may be solved for independently since they appear only in terms with $r_{i^k}^s$ and not with other $p_{i^k j}^f$'s. Taking the derivative of Equation 67, setting equal to 0, and solving for $p_{i^k j}^f$ gives:

$$p_{i^k j}^f = \sqrt[3]{\frac{2r_{i^k}^s c_{ij}}{\left(r_{i^k}^s - e_{ij}\right)\left(t_{ji^k} p_{ji^k}^p + j_{i^k j}^{trans}\right)}}$$

**Equation 68**

Equation 68 demonstrates a few interesting results. It shows that as the power of computing on the hierarchical resource becomes greater, the allowable probability of false alarm on individual modules gets smaller. Interestingly, this result holds no matter what the base, non-scalable power consumption of the individual modules themselves, only in relation to how "expensive" it is to use the hierarchical resource. In order to conserve most power, if a hierarchical resource is "expensive", its use should be protected more, but if it is "inexpensive", its use need not be protected as much. Also, for a constant environmental event rate, as a module's sample rate increases, the probability of false alarm should decrease, but should decrease asymptotically to a minimum. If more samples are taken, in other words, they should be processed with a lower probability of false alarm algorithm.

If the dependence of $p_{i^k}^n$ on $p_{i^k j}^f$ is different than inverse square, but still inverse and monotonic, these general dependency characteristics will not change, and only the amount to which $p_{i^k}^n$ is affected by a parameter variation will change. Figure 15 below shows how $p_{i^k}^n$ changes with $p_{i^k j}^f$ and $r_{i^k}^s$ for inverse, inverse square, and inverse cube

relationships between $p_{i^k}^n$ and $p_{i^k j}^f$. Parameters for these graphs were chosen to be $P_{i^k}^{Base} = 50mW$, $c_{ij} = 1x10^{-6}$, $p_{i^k j}^d = .9$, $e_{ij} = 1$, $t_{ji^k} = 50ms$, $p_{ji^k}^p = 500mW$, and $j_{i^k j}^{trans} = 250uJ$.

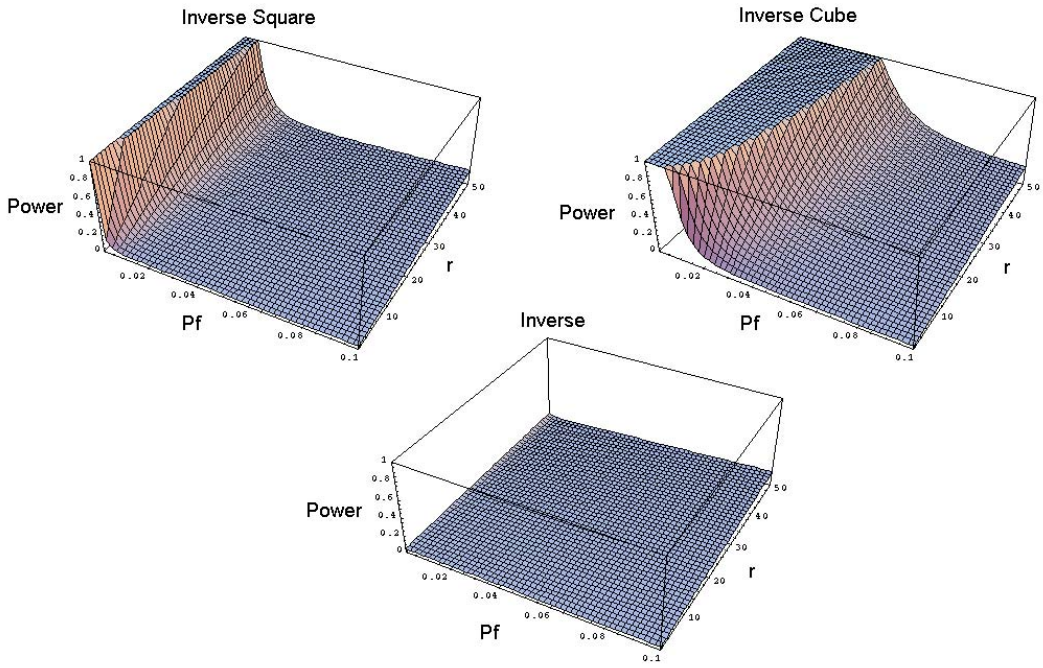


**Figure 15: Power Versus Sample Rate and Probability of False Alarm**

Since the choice of $p_{i^k j}^f$ will be discrete as opposed to continuous, Figure 15 shows that choosing the discrete $p_{i^k j}^f$ above the optimal $p_{i^k j}^f$ will give a lower total power than choosing the discrete $p_{i^k j}^f$ below the optimal $p_{i^k j}^f$.

## Optimal Module On-Time When Modules Require Transition Time and Power:

One other method of decentralized control possible in the system is that modules should power themselves down when they are not in use. If modules require some amount of transition time to power on and off, however, a module would want to remain

on if another request will be made shortly in order to conserve the transitioning power. To find the optimal on-time, the following parameter definitions are necessary.

| | |
|---|---|
| $t_i^o$ | Total time module i remains on to process and wait for more requests when its queue is empty. $t_i^o \geq t_{ij^k}, \forall j$ . |
| $j_i^n$ | Total power consumed by module i when transitioning from off to on. |
| $j_i^f$ | Total power consumed by module i when transitioning from on to off. |

**Table 5: Module Transition Power Parameters**

Assume requests occur on the following timeline:

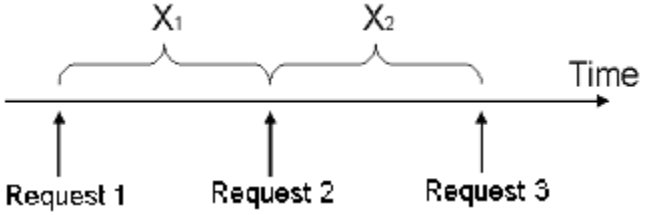


**Figure 16: Timeline of Requests**

As previously defined in Equation 25 and Equation 26, $\lambda^t_{Total,j}$ is the aggregate inter-arrival rate of requests into module j from other modules, and $M(x;\lambda) = \lambda^t_{Total,j} e^{-x\lambda^t_{Total,j}}$ is the distribution of inter-arrival times, x. Let the power associated with a request is the total power consumed while processing a request, plus the power consumed by remaining on after the request is processed, plus any transitioning power generated by the request. Consider the power associated with Request 2, P:

$$P = \begin{cases} if \quad x_1 \geq t_i^o \quad \begin{cases} if \quad x_2 \geq t_i^o \quad , \quad j_i^n + j_i^f + p^p_{ij^k} t_i^o \\ if \quad x_2 < t_i^o \quad , \quad j_i^n + p^p_{ij^k} x_2 \end{cases} \\ if \quad x_1 < t_i^o \quad \begin{cases} if \quad x_2 \geq t_i^o \quad , \quad j_i^f + p^p_{ij^k} t_i^o \\ if \quad x_2 < t_i^o \quad , \quad p^p_{ij^k} x_2 \end{cases} \end{cases}$$

**Equation 69**

Since the exponential distribution is memory-less and hence $x_1$ and $x_2$ are independent:

$$P\left(x_1 \geq t_i^o\right) = P\left(x_2 \geq t_i^o\right) = e^{-t_i^o \lambda_{Total,i}^t}$$

**Equation 70**

and

$$P\left(x_1 < t_i^o\right) = P\left(x_2 < t_i^o\right) = 1 - e^{-t_i^o \lambda_{Total,i}^t}$$

**Equation 71**

Substituting these probabilities into Equation 69, and noting that the average of the exponential distribution is $\frac{1}{\lambda_{Total,i}^t}$, the average power consumed by event two, or any other event, will be:

$$P = e^{-t_i^o \lambda_{Total,i}^t} \left( j_i^n + j_i^f + p_{ij^k}^p t_i^o + \frac{p_{ij^k}^p}{\lambda_{Total,i}^t} \left( e^{t_i^o \lambda_{Total,i}^t} - 1 \right) \right)$$

**Equation 72**

Graphing Equation 72 for $j_i^n + j_i^f = \frac{1}{2}$, $p_{ij^k}^p = 1$, and $\lambda_{Total,i}^t = 1$ gives Figure 17 below.

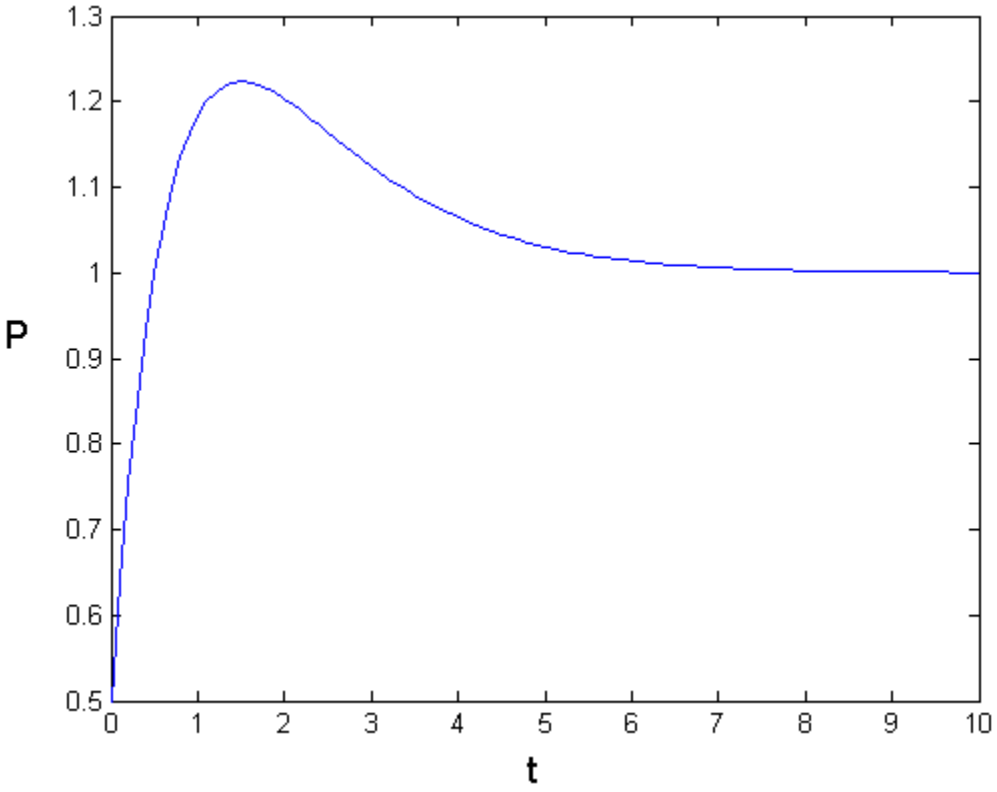


**Figure 17: Power versus turn-off time, Example 1**

Notice that for these parameters, the minimum event power is given as at a turn off time of 0. In other words, for these parameters, modules should turn off as soon as they can after processing a request. Graphing Equation 72 for $j_i^n + j_i^f = \frac{3}{2}$, $p_{ij^k}^p = 1$, and $\lambda_{Total,i}^t = 1$ gives Figure 18 below.

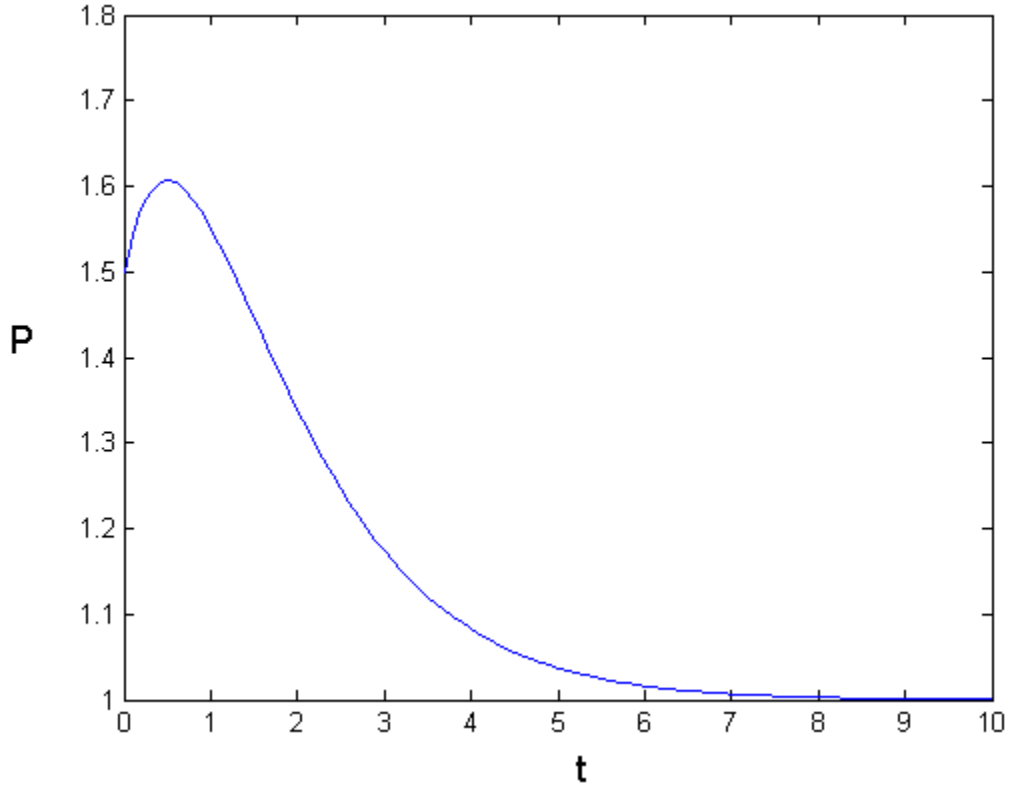


**Figure 18: Power versus turn-off time, Example 2**

As can be seen, for these parameters, the minimum power will be achieved by leaving the module on continuously.

In general, for $t_i^o \geq 0$, the graph of P will either have a positive bump and then monotonically decrease, or will simply monotonically decrease. Thus, the minimum value of P will either occur for $t_i^o = 0$ or $t_i^o \to \infty$ (or both). From Equation 72, it can be found that $P\big|_{t_o^i=0} = j_i^n + j_i^f$ and $P\big|_{t_o^i \to \infty} = \frac{p_{ij^k}^p}{\lambda_{Total,i}^t}$. Given that $t_i^o$ must be greater than some

minimum finite processing time, $t_o^i = t_p^i$, a module will thus either want to turn off immediately if $P\left(t_p^i\right) \le \frac{p_{ij^k}^p}{\lambda_{Total,i}^t}$ or remain on indefinitely.

Dynamic estimation of aggregate inter-module request rate:

The last thing needed to actually employ this time-out turn-off technique is to estimate $\lambda_{Total,i}^t$ on the individual modules. This will require keeping track of the difference between arrival times, but a simple timer can accomplish this easily. The same sort of moving average technique employed above for $e_{ij}$ is not necessarily appropriate because during long periods of no requests, $\lambda_{Total,i}^t$ will simply grow smaller and smaller, and when a request does eventually occur, the turn-off time, $t_i^o$, will be extremely long and inappropriate. The geometric average is less susceptible to extremely high outliers, and thus this average is likely a better choice. However, the arithmetic average if outliers are discounted is really what $\lambda_{Total,i}^t$ should be, so a scale factor must be assigned to the geometric average in order to compensate for this difference. The arithmetic aggregate average can be computed from:

$$a_{n+1} = \frac{n}{n+1} a_n + s_{n+1}$$

**Equation 73**

Where $a_n$ is the nth estimate of the arithmetic average and $s_n$ is the sth sample taken. The scaled geometric average can be computed from:

$$g_{n+1} = \frac{1}{c}\left(c g_n\right)^{\frac{n}{n+1}} \left(s_{n+1}\right)^{\frac{1}{n+1}}$$

**Equation 74**

Where $g_n$ is the nth estimate of the geometric average and c is the scaling factor. c is found by taking the geometric average of several independent random variables, and for the case of exponentially distributed random variables (as the inter-arrival times are assumed to be), $c \approx .561$. If the samples are taken to be the amount of time between

request arrivals, the average of these times can be computed by one of the averaging methods shown, and then $\lambda^{t}_{Total,i}$ will simply be the reciprocal of this average.

Using Equation 73 and Equation 74, Figure 19 below was generated. Figure 19 shows three different simulations for which an environmental event causes 5, 50, or 500 inter-module request rates. In other words, when an event occurs, sensor modules that detect the event will request verification from another module 5, 50, or 500 times. These numbers can also be interpreted as the duration of the event in units of inter-module requests generated. The average of the exponentially distributed inter-module requests was set at 10 time units, and periods of no events were assumed to be 1000 time units long. The final arithmetic and scaled geometric averages are shown in the titles of each graph.

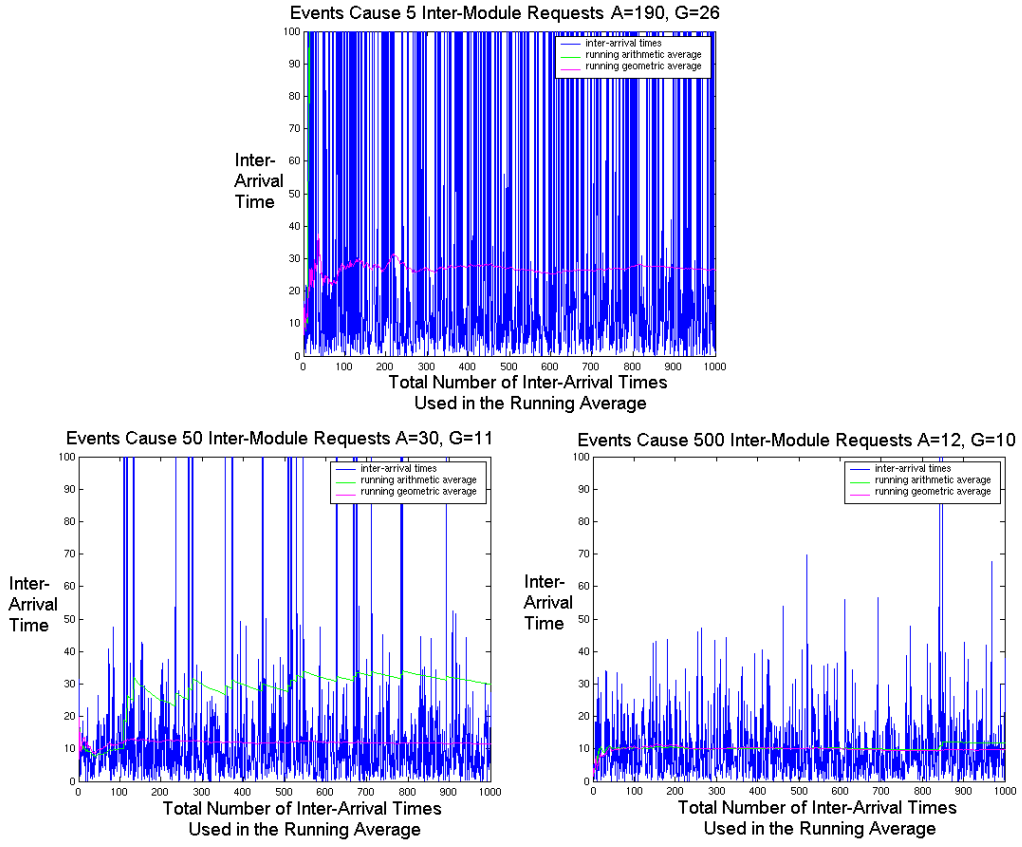


**Figure 19: Arithmetic and Geometric Running Averages**

As can be seen, the scaled geometric average is far less sensitive to long inter-arrival times even when only a few requests are generated for each event. Another way to look at this insensitivity is that it takes far fewer requests per event for the geometric

average to converge than for the arithmetic average as evidenced by Figure 20 below. Simulations also show that increasing the time between events [1000 time units] also had little effect on the geometric average, but significant effect on the arithmetic average.

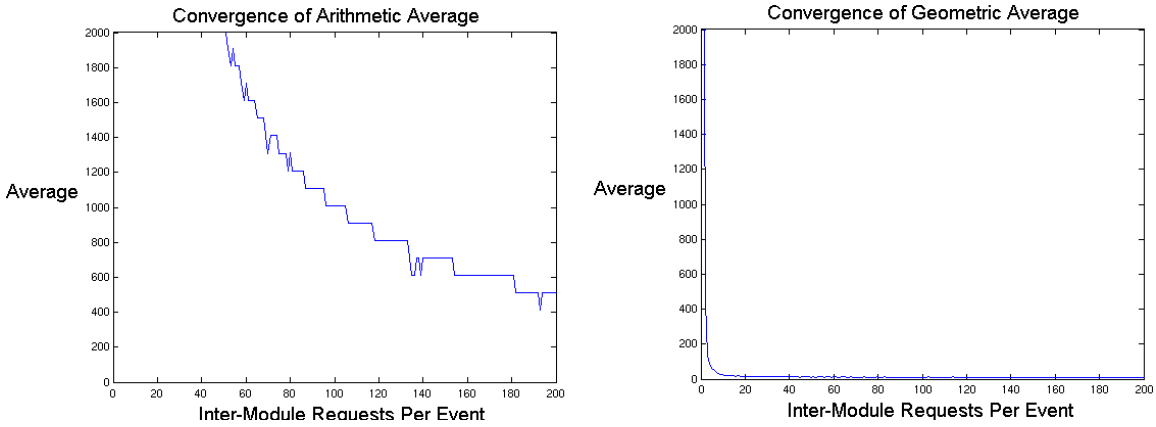


**Figure 20: Convergence of Arithmetic and Geometric Averages in Requests Per Event**

These simulations show that using the scaled geometric running average from Equation 74 will be the better of these two choices for generating estimates of $\lambda^t_{Total,i}$ on each module. The estimation of $\lambda^t_{Total,i}$ can also be derived from a moving average and outlying inter-arrival request times, signifying that no events occurred for a long time, may also be simply discarded and not included in the average if a reasonable discarding strategy is developed.

## Module mode changing based on system configuration:

Modules will change modes also depending on the configuration of the system, i.e. the other types of modules comprising the node. (Refer to the "Proposed Node Architecture" section for configuration definitions.) If the system configuration changes from $C_2$ to $M_2$, for example, one would not expect an S to continue reporting to a no longer existent GPP. Following are descriptions of mode changes that will occur for each type of module.

*S Modules:*

S's may occur in $D_1$, $M_2$, $C_1$, $C_2$, or $C_3$ nodes. They collect data from attached sensors and analyze this data with a simple algorithm (e.g. threshold detection, etc…) for

possible events. If there is an event, the S logs the data that caused the event and will then seek to send the data to a GPP for event verification.

If the S is in a $D_1$ (PS+S) node, it will simply log data without attempting to send it to any other modules, since no others exist. They may log either all of their data, or only that data in which they detect an event to have possibly occurred. In this configuration, S modules will use the highest level algorithm they have for event detection (e.g. they would choose frequency analysis over simply threshold detection). The reason for this action is in case the node is ever picked up, this un-communicated data collection can be analyzed post facto.

If an S is in an $M_2$ (PS+WNC+S) node, any detected events will be sent to the WNC which will send the data to a GPP on another node for verification. In this way, an S's data may be used by the network as a whole even if no GPP exists on board the same node as the S. In this configuration, the S's will still log their own data and events and expect to hear responses from a GPP on another node if that GPP processes its data.

If an S is in a $C_1$ (PS+WNC+$S_1$+$S_2$+(S…)) node, it will operate the same as in an $M_2$ node. The reason for this is that even though the system configuration is complex, there is no GPP on board the node to do any processing of sensor data. Essentially, the primary difference between $M_2$ and $C_1$ is that there is potentially more wireless traffic out of a $C_1$ node than an $M_2$ node.

If an S is in a $C_2$ (PS+WNC+GPP+S+(S…)) node, it will send any preliminary events and sensor data to its local GPP. It sends the data to a GPP for further analysis and verification of the event. It will still log its own data, but simply does not need to use the WNC at all to communicate with GPP's on other nodes.

If an S is in $C_3$ (PS+WNC+$GPP_1$+$GPP_2$+(S,GPP…)), it will send its events to whichever is the appropriate GPP or GPP's. The GPP's will determine between themselves how they partition the tasks of the system and tell each S from which they require data to report to them if an event occurs. Thus an S in this configuration will need a list containing the GPP's to which it needs to report each event it detects.

*GPP Modules:*

GPP's may occur in $D_2$, $M_3$, $C_2$, or $C_3$ nodes. The GPP's are essentially the "application" modules in the sense that they are the primary components that need to be reprogrammed from application to application. They are the brain of the system and can combine and verify data from the S's (from their own and possibly other nodes), as well as report to other nodes, collaborate with other nodes, and report to a user. GPP's will likely be mostly event driven in the sense that they remain dormant unless stimulated by an outside stimulus from an S or a request from another node. In other words, they will be mostly non-self-stimulating for simple applications, but they are not restricted to only this type of operation. GPP's are the most abstractly defined and flexible of all the module types.

If a GPP is in a $D_2$ (PS+GPP) node, it will have nothing to do and should shut down completely. Without any stimuli from sensors, the GPP's will have no data to compute on, and are therefore useless. The module bus connectors should remain alert in case another module is attached, but otherwise, the entire processor that the module supports can be powered down.

If a GPP is in an $M_3$ (PS+WNC+GPP) node, it can take part in distributed network computations via the WNC, verify sensor events from $M_2$ or $C_1$ nodes, and operate in its normal application supporting mode.

If a GPP is in a $C_2$ (PS+WNC+GPP+S+(S…)) node, it will operate essentially the same as if it is in $M_3$ just with potentially more request traffic coming in from the attached S modules.

If a GPP is in a $C_3$ (PS+WNC+$GPP_1$+$GPP_2$+(S,GPP…)) node, it will have to collaborate with the S modules that it requires input from in order to coordinate node operations with the other GPP's. It may also communicate with the other GPP's in the system, and carry on the normal GPP operations of communicating through the WNC with other nodes.

*WNC Modules:*

WNC's may occur in $M_1$, $M_2$, $M_3$, $C_1$, $C_2$, and $C_3$ nodes. Their operation will be mostly the same in any of these configurations as they will always route network traffic

and pass messages between modules on disparate nodes. They are essentially meant to be a way to "wire together" all of the modules in the whole system. There will be slight differences in operational necessities based on the particular potential configurations, however, so slightly more configuration-specific detail is required.

If a WNC is in an $M_1$ (PS+WNC) node, it simply routes information throughout the network. There is no data sink or source on the node, and so no interaction will occur with the WNC except by other external WNC's. In this case, the WNC's controller will simply monitor the system bus in case another module is attached.

If a WNC is in an $M_2$ (PS+WNC+S) node, it will receive messages from the S and be required to understand what to do with them. In this case, the WNC will send the event data from the S to all of the neighboring nodes with a return address of itself attached. This requires the WNC to know who its neighboring nodes are and their system configurations. If a node with a GPP receives this transmission, it will respond to the originating WNC saying that it will process the request. The WNC will send a cancellation of the processing request to any GPP enabled node that responds except for the first one so that only one GPP will process the request. (It may also send a broadcast cancellation of the request [with the servicing node address, so that the servicing node does not cancel the request] to all neighboring nodes exactly as the original request was sent.) In this way, when the S checks up on its request if it has not heard a response or acceptance of the request, it can establish a single GPP to which to address this check. If a node without a GPP receives the WNC's transmission, it will simply route the information along to all of its neighbors. These requests will have to be uniquely identified, so that routing loops don't occur, and so they can be intelligently stopped. In this configuration, the WNC is thus supporting simple distributed network computation.

If a WNC is in an $M_3$ (PS+WNC+GPP) node, it will receive messages for and transmit messages from the attached GPP. The GPP will be required to control which other nodes it wishes to communicate with and why. This is therefore an application level issue that will have to be considered each time a wireless sensor network application is built. If any distributed network computation between the GPP's on different nodes is to occur, the GPP's will have to organize it themselves. The WNC will also route all information not intended for its attached modules without disturbing them.

If a WNC is in a $C_1$ (PS+WNC+$S_1$+$S_2$+(S…)) node, it will operate exactly as it does in $M_2$. The only difference is that it may be required to handle more traffic and maintain more state when waiting for a network GPP to respond saying it will process a given S's request.

If a WNC is in a $C_2$ (PS+WNC+GPP+S+(S…)) node, the attached GPP will control the interaction of the node with other nodes. As in $M_3$, the communication is an application level issue. Again, the WNC will route all information not intended for its attached modules.

If a WNC is in a $C_3$ (PS+WNC+$GPP_1$+$GPP_2$+(S,GPP…)) node, it will operate much like in the $C_2$ case. It is left up to the multiple collocated GPP's to sort incoming traffic between themselves so tasks are not duplicated. Any of the GPP's may use the WNC to communicate with other nodes. The particular nodes that the WNC communicates with will be controlled by the GPP's, and again the WNC will route all traffic on its own.

*PS Modules:*

PS's will occur in all configurations. They are simply the power supplies to the system, and so must be universal node components. In the first prototype, they are unintelligent modules that simply deliver power to the system, but in possible future systems, their power delivery may be controlled in a more complex and fine-grained manner. They may also separate power buses to individual modules in the future which could allow them to completely power down whole sections of the node. Since these changes are unlikely to occur in the short term, nothing more will be described about the PS's.

## IMPLEMENTATION:

### *SOFTWARE STRUCTURE:*

In the MASS implementation communications bus, four of the seven OSI layers are used. A physical (Phy) layer handles each byte to be transmitted without knowing what the bytes mean. The physical layer is often given a buffer containing one complete message and the necessary information to send the message (destination, length of message, etc.). One layer higher, the data link (Link) layer takes less generic information (a destination, source, length, data, etc.) and converts it to a format the physical layer can understand. This may involve changing 16-bit words to bytes, fixing byte ordering, encoding data, adding a checksum, or a variety of other tasks. Still, the data link layer doesn't know what exactly it is transmitting. Above the data link layer, the networking (Net) layer knows how to handle specific types of data. For example, it may know that one type of data should be sent to device A on the bus, while all other types of data go to device B. Additionally, the networking layer may be aware of what device A is and what it does. It may also know that to devices A and B, it is device C. The final layer, the transport (Transport) layer, is responsible for fragmenting and reassembling messages that are too large to be sent monolithically. It will take large messages, break them up into packets, and add information in a header as to how to reassemble the packets in the destination module. Taken as a whole, this modular structure to the networking layers allows relatively simple substitutions of different data formats and busses without changing the high-level operation of the devices. It also requires each layer to be successively more intelligent and isolates each layer from the others.

In a typical example, a transport layer may be instructed to transmit a piece of sensor data. The transport layer sections the data into appropriately sized blocks and adds information to each block as to its position in the overall data. Each fragment is then given successively to the networking layer which is told the amount of data and what the data represents. The networking layer must then decide what to do with the data. It may, for example, know that this type of data should be sent to device A and that its own bus designation is device C. The networking layer sends the data, the length of the data, the source (device C), and the destination (device A) to the data link layer. The data link

layer packages the source (C), the data, and the length of the data into a new set of data (and therefore a new amount of data), then encodes and adds a checksum to the new data. The new data is then sent to the physical layer. The physical layer receives only a destination (A), a length, and a byte array of data which it transmits to the destination. Device A will process the data in the reverse order. If device A receives the data, but the data has been corrupted, the data link layer on device A will recognize the data as corrupt and discard it. In order to ensure reliable end to end communication, the networking layer on device C would have to wait for a response from the networking layer on device A. If A does not respond, C would need to attempt to send the data again.

Above the Transport layer sits the application (App) layer. The App layer is composed of three major functional units, the Request Processor (RP), Local Event Handler (LEH), and Mode Changer. The RP processes requests sent to the current module by other modules. The LEH makes requests of other modules on behalf of the user. Finally the mode changer tracks the operation of the RP and LEH and dynamically configures the module for maximum efficiency. The programmer (or user) can interface with MASS through a simple API that abstracts away the complications of communicating with other modules. Figure 21 graphically depicts the software structure.

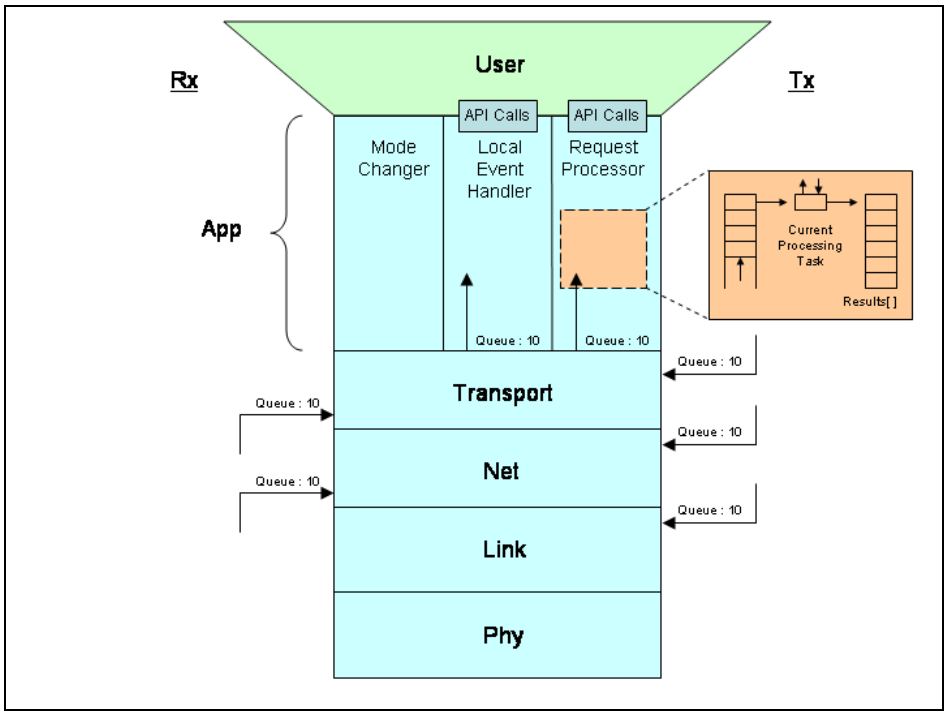


**Figure 21: Layered software structure**

*SIMULATION:*

A simulation of the architecture was undergone following the mathematical analysis in order to develop an appropriate software structure for the decentralized system. Since the simulation was built using a high level simulation package, ideas could be tested quickly without worry of wasted time down dead-end paths that could have occurred if the simulation stage was skipped. The simulation was built as a large finite state machine with both parallel and serial state transitions. A method of dynamic self-addressing was developed and tested during the simulations, as well as the priority queue based method of requests and responses throughout the node. A description of the simulation details will not be undertaken because the following section describing the final software implementation supercedes it. Figure 22 and Figure 23 show screen shots from the simulation.

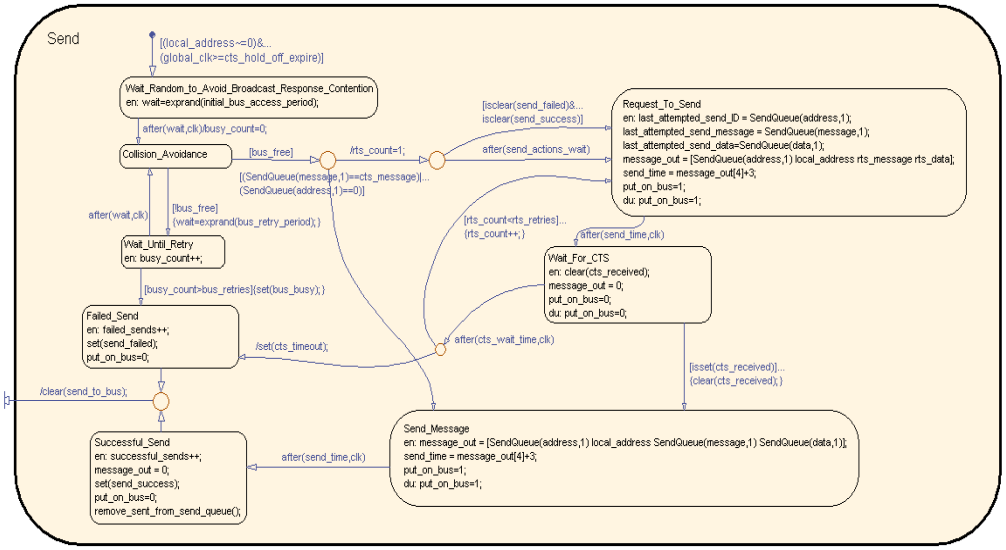


**Figure 22: Example meta-state in the simulation physical layer**

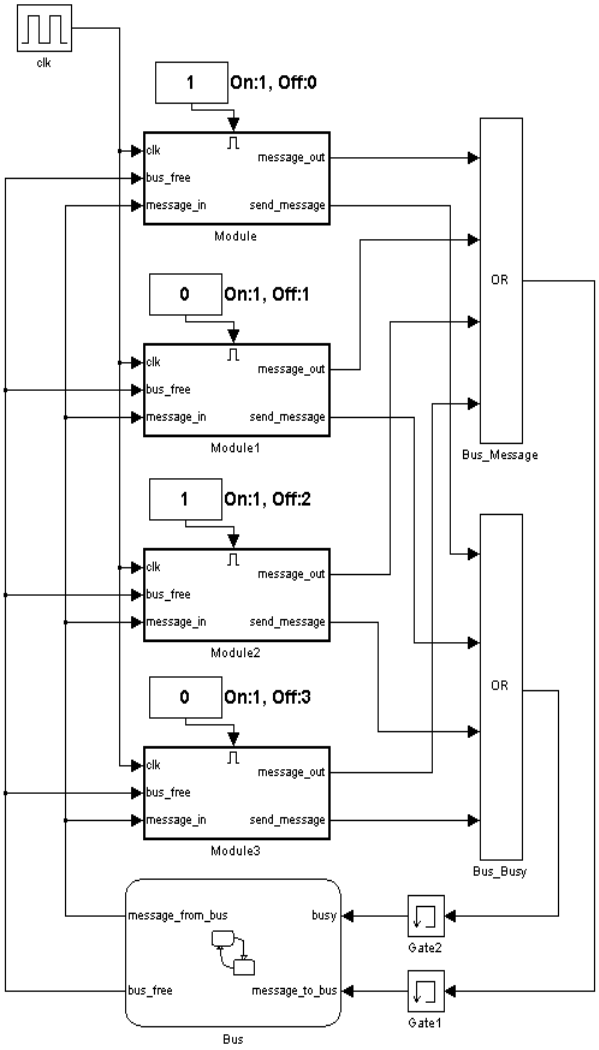


**Figure 23: Top level view of the simulation system**

## *HARDWARE AND SOFTWARE PROTOTYPE:*

### Hardware:

The first iteration of MASS was implemented on an 8051 prototyping board shown in Figure 24 below. The Keil uVision-II 8051 compiler was used to generate the code for the microcontroller. The software was developed on uC/OS-II. The 8051 prototyping boards are capable of stacking together on a common 80-pin bus. The modules were configured to share an I2C bus for communication. As of this writing, MASS has also been ported to an ARM7 platform, but additional testing is required.

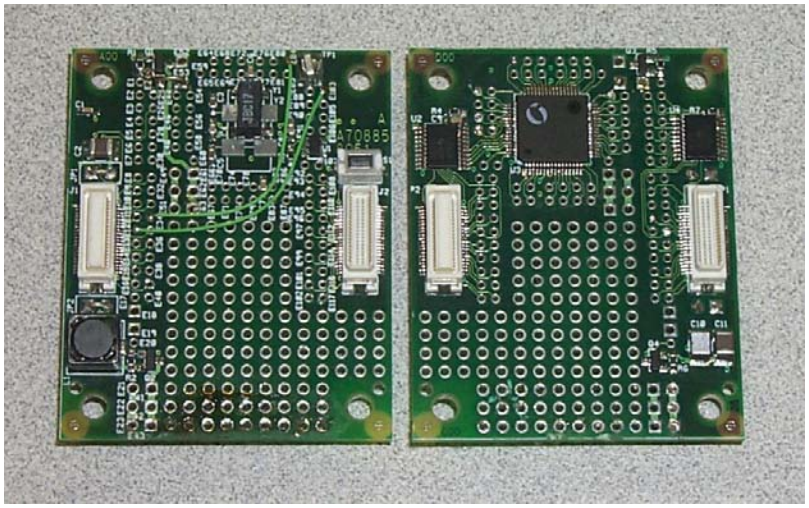

**Figure 24: Prototyping board used for implementation**

The best available bus for this hardware was the onboard I2C bus. I2C is a multi-drop, multi-master bus supporting bit rates of 100kbps or higher. In the case of this hardware, 100kbps was used. Unfortunately, the 8051 links the I2C bit rate to the internal clock of the microprocessor, which means the processor cannot be completely asleep while still remaining able to respond to bus activity. This is further complicated by the clock stretching features of the I2C bus. If the I2C hardware detects bus communication that is too fast for it to handle, it effectively slows down the entire bus to a manageable speed. In the case of the 8051, when any processor on the bus was asleep, the entire bus would slow down to about 8kbps.

## Messaging in MASS:

Communication between modules in MASS is accomplished via messages with a pre-determined structure and finite set of types. Messages in MASS have the following structure:

```
struct Message {
     INT8U to;
     INT8U from;
     INT8U flags;
     INT8S prio;
     INT8U msgID;
     INT8U cmd;
     INT16U dataLength;
     INT8U* dataPtr;};
```

The `to` and `from` fields are the destination and source of the message, respectively. The `flags` field is used for fragmentation, the `prio` field denotes the priority of the message if it is relevant, `msgID` serves to uniquely identify a message within a module, `cmd` indicates the type of the message, and `dataLength` specifies the amount of data contained in the data pointer, `dataPtr`.

*Message Types:*

There are 13 different message types in MASS which fall into four broad classes.

- Class 1 messages: `IDBroadcast` or `IDContention` messages that are use for address determination and network stabilization between modules.
- Class 2 messages: messages which can carry significant data payloads, such as `ProcResult` and `FullProcReq`.
- Class 3 messages: single-stage processing request and response messages, including `ProcReq`, `ProcReqAccept`, `ReqInQ`, `ReqProcessing`, `QFull`, `ReqBump`, and `ChangePriority`.
- Class 4 messages: two-stage processing request and response messages, including `ReqForFullReq` and `FullReqAccept`. `ProcResultAck` also falls in this class even though it is common to single and dual-stage processing requests.

The purpose and effect of each of these message types is discussed in detail below:

- IDBroadcast: A broadcast message identifying a module to all other modules in the node. This message contains the module's address and type information.
- IDContention: A broadcast message used to contest a module that has just identified itself with an address that is already in use.
- ProcReq (Processing Request): This message is used by one module (the requestee) to request processing time on another module (requested module). This message is only a stub; it does not contain the actual data.
- ProcReqAccept (Processing Request Accept): Sent in response to a Processing Request, this message indicates that the processing request was accepted.

- ReqInQ (Request in Queue): Sent in response to a Processing Request, this message indicates that the processing request had already been received and accepted.
- ReqProcessing (Request Processing): Sent in response to a Processing Request, this message indicates that the processing request was already accepted and is currently being processed.
- QFull (Queue Full): Sent in response to a Processing Request, this message indicates that the processing request was denied.
- ReqBumped (Request Bumped): This message is sent to the module that sent a processing request when that request is bumped out of the requested module's queue.
- ChangePriority: Used to request that the requested module change the priority of a processing request. It can also be used to cancel a processing request.
- ReqForFullReq (Request for Full Request): Sent by the requested module to the requestee when a processing request reaches the top of the requested module's queue and is ready to be processed.
- FullProcReq (Full Processing Request): Sent in response to a Request For Full Request, this message contains the actual data to be processed and instructions on how to process it, if any.
- FullReqAccept (Full Request Accept): Sent in response to a Full Processing Request, this indicates that the Full Processing Request was received correctly.
- ProcResult (Processing Result): The result of a processing the Full Processing Request.
- ProcResultAck (Processing Result Ack): Indicates that the processing result was received correctly.

*Example Messages:*

Below are two example messages with explanations which illustrate the potential uses of each message field.

- IDBroadcast
    - to – 0x00 (Indicates this message is a broadcast)

    - from – 0x80 (The bus address of the module sending the message)
    - flags – 0x00 (Used by transport layer to fragment messages)
    - prio – 0x10 (A signed number representing the priority of the message)
    - msgID – 0x00 (An increasing count of the number of messages this module has sent – unused for IDBroadcast)
    - cmd – 0x02 (The number used to identify IDBroadcast messages)
    - dataLength – 0x0002 (Number of bytes of data in the dataPtr)
    - dataPtr (Address at which the data resides in memory)
        - {0x43, 0x40} – A random number in the first byte and a number representing the type of the module in the second byte
- FullProcReq
    - to – 0x74 (Destination is address 0x74)
    - from – 0x80 (The bus address of the module sending the message)
    - flags – 0x00 (Used by transport layer to fragment messages)
    - prio – 0x20 (A signed number representing the priority of the message)
    - msgID – 0x05 (This module has sent 5 processing requests)
    - cmd – 0x0C (The number used to identify FullProcReq messages)
    - dataLength – 0x0004 (Number of bytes of data in the dataPtr)
    - dataPtr (Address at which the data resides in memory)
        - {0x18, 0x75, 0x38, 0x42} – The actual data associated with the message, which is only meaningful to the user. MASS does not interpret or even look at the data.

In both cases, the entire message would be transmitted as string of bytes followed by a calculated checksum.

### Task Overview:

The 13 different messages are use to accomplish a variety of tasks including but not limited to address generation and resolution, sending processing requests and receiving the responses, and checking the status of outstanding processing requests.

These tasks are described in order to demonstrate one or more uses for each message type.

*Address Generation and Resolution:*

As soon as a module is connected to the bus, it generates an address and sends an IDBroadcast message to all modules containing that address. Addresses are generated using the random number generator in `stdlib.h` which is seeded by reading an analog to digital converter which is not connected to any sensor. Addresses generated are checked to make sure that they are locally unique, that is the module generating the address has not already received an IDBroadcast message from a module with that address. If a module receives and IDBroadcast message with the same address as it's own, it sends an IDContention message to all modules containing its address (which is the contended address). Upon receiving this IDContention message, the module which originally sent the IDBroadcast message containing the contended address will generate a new address and send another IDBroadcast message with this new address. This process continues until the module is able to successfully generate a unique address which no other module contends.

*Processing Request/Response:*

Processing requests are generated by the user through calls to the Local Event Handler. This results in either a ProcReq or a FullProcReq message being sent depending on the size of the request data. The module that receives this Processing Request message responds with a ProcReqAccept or FullReqAccept message if there is space on the request queue for the request, or a QFull message if the request queue is full. If the request is already in the processing queue or if the request is currently processing a ReqInQ or ReqProcessing message may be returned instead. Once the processing module is ready to process the request it sends a ReqForFullReq message if it does not already have the data for the request. The requesting module then responds with a FullProcReq. Once the request is processed, the processing module sends a ProcResult message to the requesting module. The requesting module then acknowledges the result with a ProcResultAck message completing the transaction.

*Checking Status of Processing Requests, Modifying Priorities, etc.:*

While waiting for the result of a processing request, the requesting module may check the status of the request by sending a ProcReq message with the same priority and message ID as the request. The processing module will then respond with a ReqInQ or ReqProcessing message if the request is already in the queue or processing. If the processing module does not have a copy of the request locally, it will try to insert it into the queue. This will result in either a QFull or ProcReqAccept message depending on whether or not the insertion is successful. If the requesting module wants to modify the priority of the message based on the results of the status request, it can send a ChangePriority message with the new priority. If at any point the processing module has to discard a processing request to make room on the queue for a higher priority request it will notify the module which sent the bumped request with a ReqBumped message.

## Future Directions:

Additional functionality will be added to MASS in the future, this section describes some of the additional features that will be added..

*Encrypted Communications:*

The ability to encrypt inter-module communications will be added in future versions of MASS. This encryption will occur in the Phy or Link layers and is intended to obscure the messages generated from within MASS to prevent eavesdroppers from gaining information regarding the internal state of a module. The encryption of communication channels in combination with hardware safeguards which prevent attackers from reading ROM and RAM will make it difficult for malicious parties to determine information about a particular MASS module or node. This is not specifically intended as a replacement for user-layer encryption of inter-node communication.

*Hardware Data Sheets:*

Currently the Net layer includes facilities for exchanging data sheets between modules using IDBroadcast messages. The currently unspecified data sheet format will be replaced with an IEEE 1451.2 compliant data sheet format.

*Multiple Bus Architecture:*

Currently a single bus is used for both control messages and data related to processing requests. A second bus will be added to handle data and large messages to prevent them from tying up the control bus for long periods of time, which may degrade performance. Control messages will then be modified to specify which channel on the data bus to receive the data associated with that message on, as opposed to transmitting it over the control bus. The new bus will most likely have a higher data rate than the current $I^2C$ bus, and have multiple channels to allow multiple simultaneous transfers.

*Alternate Processor Architectures:*

Porting MASS to additional architectures is also a future goal, current projects include the ARM7 and TI MSP430.

*Additional Module Types:*

The development of a Camera module is currently underway for image recognition research, as well as a General Purpose Processing module to expand the capability of nodes to perform local computation. The GPP module will most likely use a powerful general purpose processor such as an ARM9 or PXA255 running Linux controlled by an 8051. The 8051 will handle the inter-module messaging and turn the GPP on and off as needed.

## MASS Documentation by Layer:

*Global Data:*

There are several files that contain data which is accessed or exported to a number of layers. The first of these is `global.h` which contains compiler directives and macros, as well as defining the types of messages in the system, the different types of modules that may exist in the system, some timing parameters, task priorities for all MASS tasks, and debugging code for turning individual layers on and off. It also contains the definition of an important structure, the “Message” structure, which defines the format of all inter-module messages.

The `user.h` file contains configuration parameters that may be adjusted by the user, as well as prototypes for API functions. Each configuration parameter and function prototype is documented and will be covered in additional detail in the following sections.

*Priority Queues:*

Most communication between layers in MASS occurs via Prioritized Queues. The Priority Queue structure provides thread-safe prioritized queuing and dequeuing of pointers to structures. Priority Queues have a fixed size which is determined at creation time. If the queue is full, a pointer can only be inserted if it is a higher or equal in priority than the lowest priority pointer on the queue. In the event of a tie between the priorities of any pointer on the queue or the pointer to be inserted, the oldest request is removed.

Priority Queues are fairly heavily genericized and non-generic operations are performed via function pointers in the queue structure. These function pointers can be assigned to user supplied functions to handle additional data types. The two functions that may be changed by reassigning their function pointers are the comparison function which is used to order objects on the queue, and the extract priority function which determines the priority of an object.

Priority Queues are available to the user whenever MASS is included in the system. The following functions are available for Priority Queues:

- `PQCreate`
- `PQPost`
- `PQPend`
- `PQRemove`
- `PQGetRemove`
- `PQStopDequeue`
- `PQStartDequeue`

- `PQCreate` creates a priority queue given a pointer to memory allocated by the user in which to store pointers to data objects, the number of data objects that can be stored in this space, the size of each data object, and the comparison function to be used to order the pointers on the queue.

- `PQPost` allows the prioritized insertion of a data object onto a queue. It requires a pointer to the queue to insert the object into, and the number of ticks to wait before timing out.
- `PQPend` allows the highest priority object on the queue to be returned, and blocks if the queue is empty. It also takes a number of ticks to wait before timing out.
- `PQRemove` removes all objects on the queue matching a specified pattern, determined by a pattern, pattern length, and offset into the object at which it should be found. It returns the number of objects deleted. It also takes a parameter which specifies how many ticks to wait to gain access to the queue before timing out.
- `PQGetRemove` finds the first object on the queue matching a specified pattern, also determined by a pattern, pattern length, and offset into the object at which it should be found. Upon finding a match it removes it from the queue and returns it to the user. This means that `PQGetRemove` removes at most one object from the queue each time it is called. It also takes a parameter which specifies how many ticks to wait to gain access to the queue before timing out.
- `PQStartDequeue` and `PQStopDequeue`, start and stop the removal of elements from the queue via `PQPend`. When dequeuing is turned off, no `PQPend` operation will succeed, however unless the pend operation times out, it will proceed as usual once dequeuing is turned back on.

Process flows for the non-trivial Priority Queue operations follow in Figure 25 through Figure 27:

Priority Queue: Post

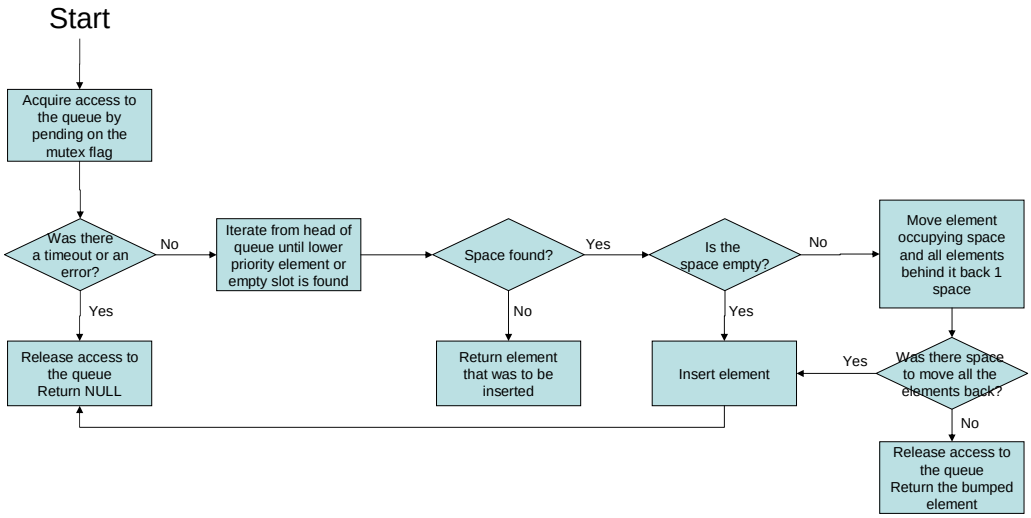


**Figure 25: Priority Queue Post process flow**

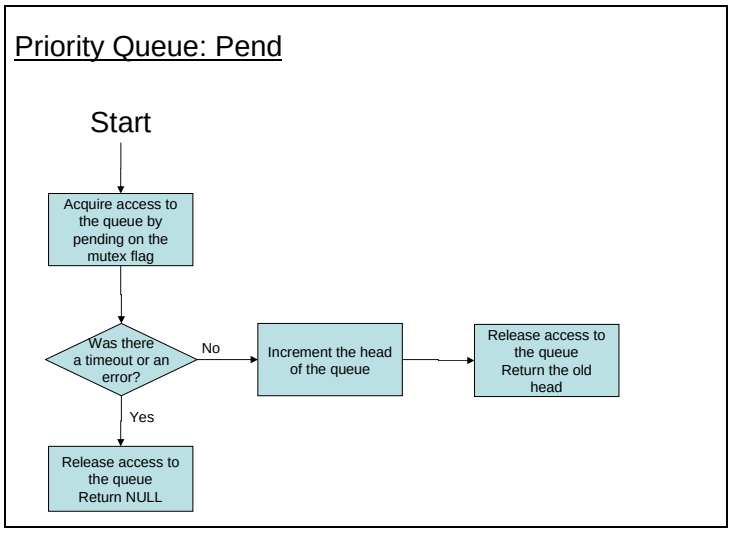


**Figure 26: Priority Queue Pend process flow**

Priority Queue: GetRemove

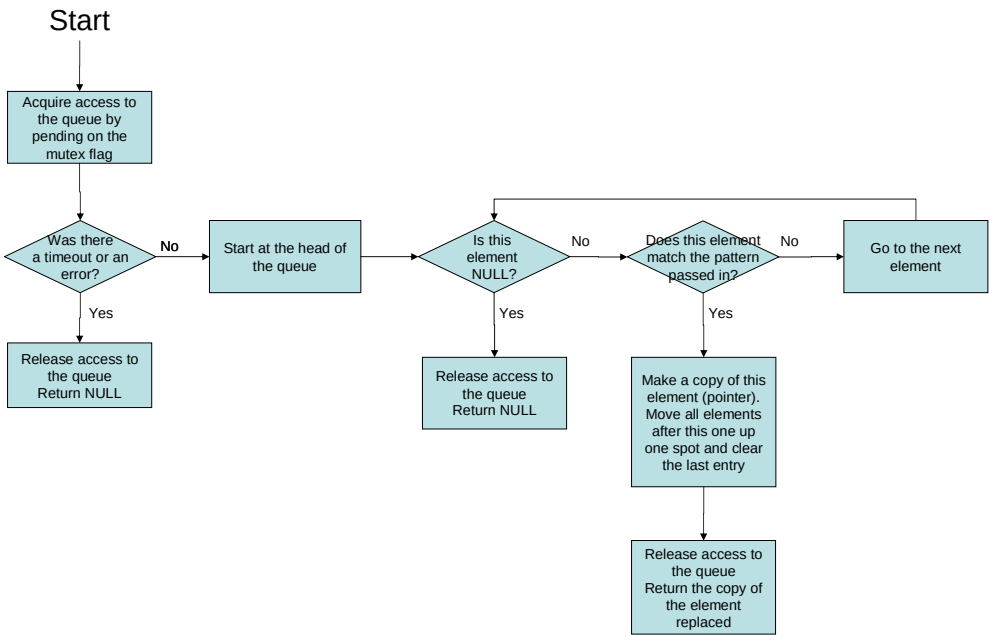


**Figure 27: Priority Queue GetRemove process flow**

*Phy:*

The Phy layer handles the actual reliable transmission of individual bytes across the I2C bus. In transmit mode, it initiates a transmission, sends the entire content of a message, sends a checksum, and then terminates the transmission. The Phy layer also handles collisions between transmissions. When two transmissions collide, the I2C hardware arbitration allows one message to be completed uninterrupted while the sender of the second message recognizes the collision. The loser of the collision is able to receive the message that won arbitration if necessary and wait until the bus is free to try to send again.

In receive mode, the receiver recognizes the start of a transmission and whether the message is for it or not. If not, the receiver simply waits for the end of the message before it starts listening again. If the message is for the receiver, it receives and acknowledges each byte of data while calculating the checksum of the message. If the received checksum matches the calculated checksum, the message is passed up to the Link layer. If not, the message is discarded.

In the event that the Phy layer cannot secure enough memory to store the incoming message, incoming bytes will be NACKed until memory is available.

The Phy layer can only buffer one message to transmit. Once it has a message to transmit, it will reject requests to send anything else. For this reason, the Link layer keeps track of the state of the Phy layer and only allows it to manage one message at a time.

The process flow for the Phy layer is shown below in Figure 28 below.

Phy Layer

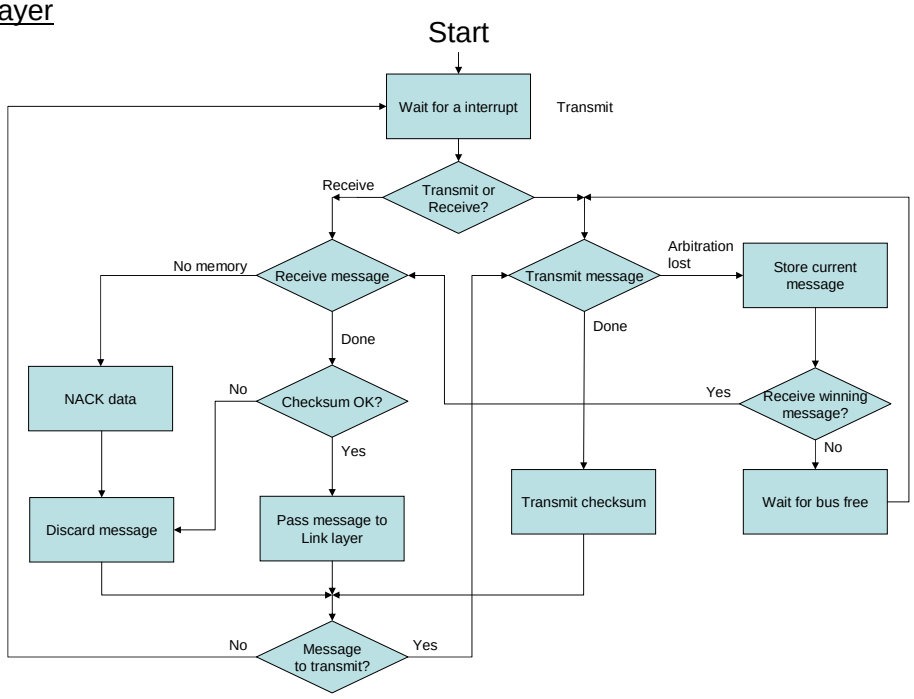


**Figure 28: Phy layer process flow**

*Link:*

The Link layer will show its use in future implementations of MASS. Currently, the Link layer checks the integrity of messages and manages the Phy layer. Messages that do not pass the integrity test are discarded. Once the Phy layer is given a message, the Link layer shuts down until the Phy layer is free.

In the future, the Link layer will service two more vital roles. First, the Link layer will manage multiple Phy layers to aid in data throughput. There will likely be at least two busses: a low data rate control bus (like I2C) and a higher data rate data bus (like an asynchronous serial or parallel bus). In order to manage communication across multiple

busses, the Link layer will also implement RTS/CTS messages to request time on the non-control bus or busses with other modules.

Process flows for the receive and transmit side of the Link layer follow in Figure 29 and Figure 30.

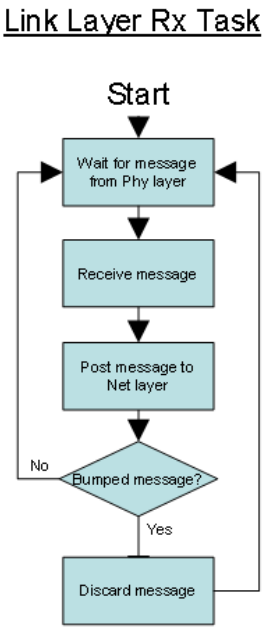


**Figure 29: Link layer receive task process flow**

Link Layer Tx Task

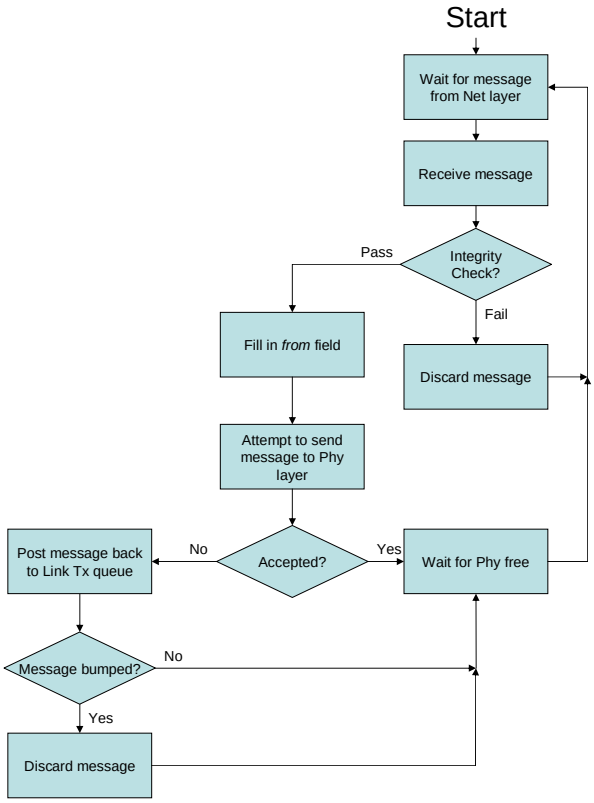


**Figure 30: Link layer transmit task process flow**

*Net:*

The Net layer has two main tasks, to inform modules of each other's existence, and to keep track of the other modules in the system. The Net layer also has provisions for returning some relevant data about the state of the system to other interested layers. The Net layer deals with class one messages, that is ID Contention and ID Broadcast messages.

There are two tasks in the Net layer, one to deal with messages received from other modules (the Rx task), and one to deal with messages to be sent to other modules (the Tx task). The Tx task handles the addressing of outgoing messages (filling in the "to" and "from" fields). If a message is not addressed to a particular module, or specified to be a broadcast, the default routing scheme is used by the Net Tx task to address it. As of the writing of this document, the default routing scheme is to send messages to the first available General Purpose Processor (GPP), and if no GPP is present to route them to the first available Wireless Network Connector (WNC). In the future, the default routing

scheme will be configurable. The Net Tx task also handles setting the “from” field of outbound messages.

The Net Rx task performs most of the work in the Net layer. It is responsible for sending heartbeat messages to other modules, and keeping track of heartbeat messages it receives to determine when modules enter and leave the system. It also performs address determination and handles address conflicts. All class 1 messages originate in the Net layer and are consumed by it as well. Class 2, 3, and 4 messages received from other modules are either posted to the transport layer via the transportRxQ, or if the transport layer is not present, posted directly to the appropriate application layer task. All processing request, full processing request, change priority, and processing result acknowledge messages are sent to the Request Processor. All other messages are sent to the Local Event Handler.

ID Broadcast messages sent by the net layer include one byte of random padding in the data pointer to guard against two modules sending identical messages. If this situation occurred and the transmissions collided, it would be unclear as to which message was successfully transmitted as $I^2C$ resolves collisions by giving arbitration to the message with the highest binary value. Without the random padding, the messages would have the same value. The random padding significantly reduces the likelihood of this occurring, though if both modules select the same random byte and have the same type, an unresolved collision situation is still possible.

The Net layer also handles data sheet management. Data sheets are sent and received via unicast IDBroadcast messages. Data sheets are descriptions of modules that can be passed around the system to give it a greater self-awareness. For example, data sheets may contain information on the type of information collected, the available processing algorithms resident on a module, tunable parameters of a particular resource, etc… No specific data sheet format is specified, but the capability was created with the IEEE standard 1451 in mind.

Process flows for the Net Tx and Rx tasks follow in Figure 31 and Figure 32.

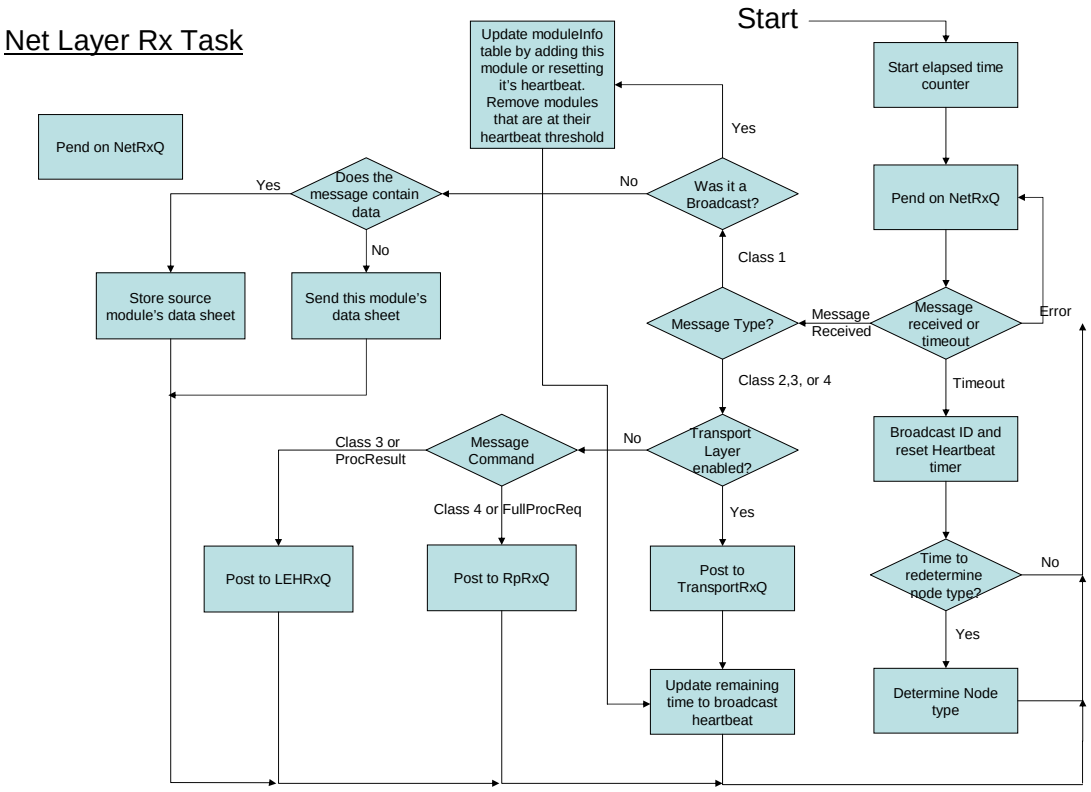


**Figure 31: Net layer receive task process flow**

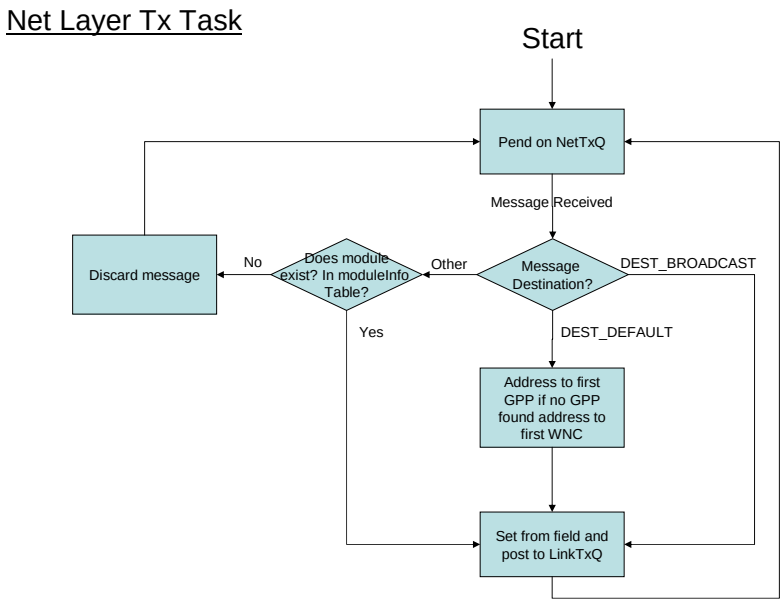


**Figure 32: Net layer transmit task process flow**

*Transport:*

The transport layer handles fragmentation of large messages to avoid tying up the bus for long periods of time. The transport layer is an optional component of MASS and may be compiled out by not defining TRANSPORT_LAYER in global.h.

When the transport layer is present it breaks messages whose total size is greater than LARGE_MESSAGE into smaller messages of size LARGE_MESSAGE, plus a final message which may be smaller. This parameter is user configurable and can be found in user.h. These smaller messages are then reassembled into the original large message at the destination. MASS's fragmentation scheme is relatively simple as of the writing of this document and has some restrictions. The first fragment of a message must be received before any subsequent fragments will be handled, if the first fragment of a message has not yet been received, later fragments will be dropped. After the first fragment is received, later fragments may arrive in any order. Fragment sizes must be the same between modules, this restriction may be removed at a later point in time.

Fragmentation works by prepending the data payload of each message with the total message size and the offset into that total amount of data at which the data contained in the current message resides. In order to preserve the isolation of the transport layer from other layers, the data payload from a message that is to be fragmented is copied one fragment at a time into additional messages and then sent as opposed to simply passing in a pointer to the point in the original data to begin copying from. The latter approach would result in the Phy layer freeing pointers which did not represent actual memory allocations, thus corrupting the memory pool. The alternative was to have the Phy layer have some knowledge of message fragmentation, which was deemed to be undesirable in that it introduced a dependency between otherwise unconnected layers. The copying of data from the original large message into the smaller messages results in the use of approximately twice as much memory as the original large message occupies. The exact amount of memory required to send a message of size N bytes is:

$$2(N-M)+M\left(\frac{N}{LARGE\_MESSAGE-M}\right)$$

Where M is the size of a message structure, 11 bytes in the current implementation.

The transport layer has two tasks, one to deal with messages received from other modules (the Rx task), and one to deal with messages to be sent to other modules (the Tx task). The Tx task performs fragmentation and sends messages out as they are created. The Rx task performs fragment reassembly by maintaining a linked list of incomplete fragmented messages. Each fragmented message on the list has a lifetime assigned when the first fragment is created, this lifetime is defined by `MESSAGE_FRAGMENT_LIFETIME_MS`. If all fragments of a message are not received within this period of time, the message is discarded. Every `FRAGMENT_CLEANUP_INTERVAL_MS` ms the fragmented message list is swept and all expired messages on it discarded. During fragment reassembly, the list is examined and expired messages deleted, the `FRAGMENT_CLEANUP_INTERVAL_MS` serves only to ensure that expired fragments are removed even if no fragmented messages are received after the expiration of a message to initiate fragment cleanup. When a message is completely reassembled, it is passed up to the appropriate application layer task.

Other configurable parameters in the Transport layer include the size at which to fragment messages and the size at which to send data as a Processing request followed by Full Processing Request rather than a single Full Processing Request. Messages under `SHORT_MESSAGE` bytes are sent as a single Full Processing Request. Messages over this size are sent as a Processing Request with no data, and then a Full Processing Request is sent when the module that is processing the request requests it. Messages over `LONG_MESSAGE` bytes are fragmented, while those under this size are not. Both these parameters are found in `user.h`. Finally, the size of the transport queues can be adjusted, the value of `TRANSPORT_QUEUE_LENGTH` indicates how many messages can be stored on each of the Transport layer queues.

Additionally, the transport layer may be disabled entirely by not defining `TRANSPORT_LAYER` in `global.h`. If the transport layer is disabled, no fragmentation is performed and the Net layer interacts directly with the App layer. This saves a significant amount of ROM as well as quite a bit of RAM.

Process flows for the Transport Tx and Rx tasks follow in Figure 33 and Figure 34.

Transport Layer Tx Task

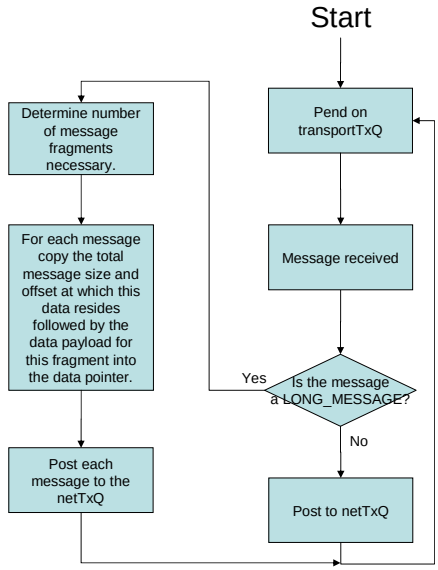


**Figure 33: Transport layer transmit task process flow**

Transport Layer Rx Task

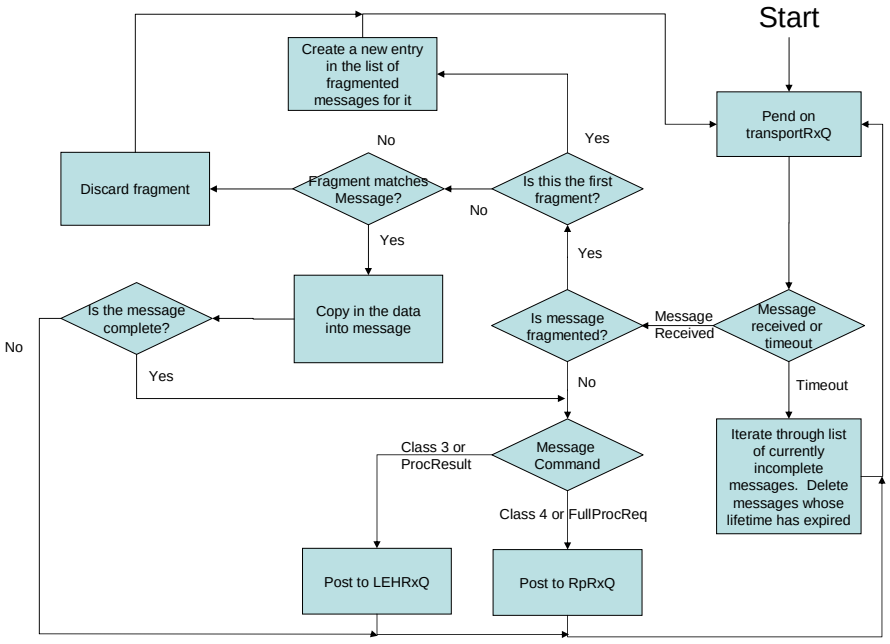


**Figure 34: Transport layer receive task process flow**

*App: Local Event Handler:*

The Local Event Handler (LEH) is responsible for sending requests from the user to other modules and returning the responses to those requests to the user. The user gives

the request to the LEH by calling `Send` with the appropriate data. The LEH stores the request and begins the process of sending the request to another module. The LEH will try up to `PROC_RQST_RETRIES` times to send the request, at an interval of `PROC_RQST_TIMEOUT` milliseconds. If the request is not accepted, it is dropped. Once the request is accepted, the LEH will continue to check the status of the request on an interval of `STATUS_TIMEOUT` milliseconds. If the LEH finds that the request has been removed from the requested module's queue or is informed by the other module that the request has been bumped, the LEH will wait `REJECTED_TIMEOUT` milliseconds and begin the process again. If at any point the LEH finds that a request has been rejected more than `REJECTED_RETRIES` times, the request is dropped.

Once a request is accepted by another module, the LEH waits until it receives a Request For Full Request from the other module. The LEH then sends the complete message to the other module. It will wait `FULL_RQST_ACCPT_TIMEOUT` milliseconds for a Full Request Accept from the other module before resending the message. If the LEH tries to send the Full Request `FULL_RQST_RETRIES` times and fails, the request is dropped.

Once the Full Request Accept is received, the LEH goes back to checking the status of the message until the Processing Result is received. The LEH responds to all Processing Results by sending a Processing Result Acknowledge back to the other module. Processing Results are checked against existing requests to find the matching request. If a match is found, the result is passed to the user and the request is deleted. If no match is found, the result is assumed to be erroneous and is dropped. The user calls `WaitForResult` to receive the result of the request.

Requests that have less than `SMALL_REQUEST_PAYLOAD` bytes of data, the LEH moves directly to sending a Full Request, skipping the Processing Request and wait for a Request For Full Request. Separating requests into short requests and long requests eliminates overhead for short requests that can be sent multiple times if necessary without impacting performance.

In addition to sending and tracking request and receiving results, the LEH also allows the user to query the state of requests, delete requests, and change request priorities. Each time the user calls `Send`, the user is given a 16-bit key to their request.

This key can be used to get the state of the request by calling `GetStatus`, or find out how many times a request has been rejected by calling `GetRejections`. The user can also use the key to change the priority of the message by calling `ChangeRequestPriority` with a new priority, or remove the request entirely by specifying `DELETE_PRIORITY` as the new priority.

The LEH also has the ability to send and receive broadcasts. To send a broadcast, the user calls `Send` and specifies the destination as `DEST_BROADCAST`. Broadcasts do not generate any results and are sent only once before being discarded. If the LEH handler receives a broadcast, it is passed directly to the user as if it were a Processing Result.

The process flow for the Local Event Handler follows in Figure 35.

Local Event Handler

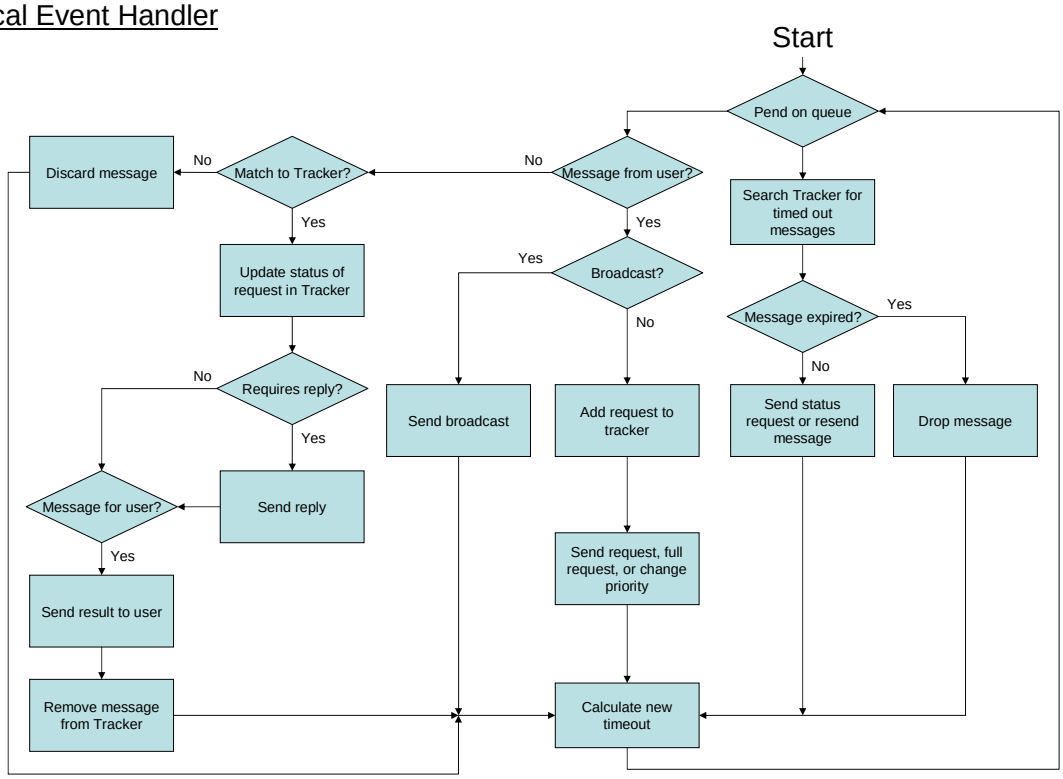


**Figure 35: Local event handler process flow**

*App: Request Processor:*

The Request Processor is the "server" side of the MASS architecture. This task handles processing requests from other tasks. It maintains a prioritized list of processing

requests from other tasks and calls a user provided function to process these requests. When requests are complete it sends the results to the requesting module.

The Request Processor consists of two tasks, the Request Processor itself which resides in the Application layer, and the User layer task which actually processes requests by calling a function provided by the user when MASS is started. The Request Processor handles prioritizing requests, storing and sending results, and responding to status inquiries or modifications to existing requests from remote modules. The User layer task simply needs to receive requests one at a time from the Request Processor, processes them, and return the results to the Request Processor.

The request processing task can be viewed as two major components. The first handles management of the queue of processing requests and controlling the User layer task. The second handles getting the results of processing requests to the requesting modules.

There are three incoming messages which can change the state of the processing request queue, and depending on the status of the request a response may or may not be generated. These messages are Processing Request, Full Processing Request, and Change Priority. There are two methods of injecting requests into the processing request queue, either all at once, or in two stages, first the request then the data associated with it. In the first scenario, a Full Processing Request message is received by the Request Processor which contains identifying information about the request (a request ID number and the priority of the request), as well as the data to be processed. In the second scenario a Processing Request message which contains no data is sent first, and when the Request Processor is ready to process that request, it sends a Request For Full Request message to the module which originated the request. At this point the module which owns the request sends a Full Processing Request message which contains the data to be processed.

When a Full Processing Request is received, the processing queue is checked to see if a request already exists from the node sending the Full Processing Request with the same request ID number. If so, but the request has no data associated with it, the data is copied in from the message, in this case, no acknowledgment of the message is sent. If the request is not found on the processing queue, the requesting module and message ID for the current processing request is checked to see if it is the same as the message

received. If so, and the currently processing request already has data to process, a Request Processing message is sent to the sender of the Full Processing Request, however if the currently processing request has no data, the data from the message is copied in and a Full Request Accept message is sent. In the case that no request matching the Full Processing Request is found on the processing queue or as the currently processing request, the Request Processor attempts to add it to the processing request queue. This will result in one of two outcomes, either the insertion is successful and Full Request Accept message is returned, or the insertion is unsuccessful and a Queue Full message is returned. In addition if any message was bumped from the processing request queue to insert the request received, the module which sent that request is sent a Request Bumped message.

Processing Requests are handled in a similar fashion with two exceptions. When a duplicate request is found in the processing queue, a Request in Queue message is returned and when a duplicate message is found to be processing, a Request Processing message is returned.

Change Priority messages generate similar responses with two exceptions. When a Change Priority message is received for a request that does not exist, the request referred to by the message is added to the processing request queue and a Processing Request Accept is returned. If the request is found on the processing queue, then the priority is changed as specified by the message and a Processing Request Accept is also returned.

Returning the results of processing requests to the modules which requested them is accomplished in the following fashion. When a processing request has been processed, a result structure is created and stored in the results array by the User layer request processing task and a Processing Result message is placed on the message queue for the Request Processor. This result structure consists of a Processing Result message, a timeout field that specifies when the result needs to be (re)sent, and a counter field that tracks how many times this result has been sent. Upon receiving the Processing Result message, the request processor scans the results array and sends all results whose timers have expired, including the result just placed in the array by the User layer task. If the Processing result message is acknowledged, the result is removed from the results array.

The message will also be removed after it has been sent a number of times equal to RESULT_SEND_ATTEMPTS. Results are resent every RESULT_RESEND_INTERVAL_MS. The results array can store information about RESULT_ARRAY_ENTRIES results and is prioritized in that a higher priority result will bump a lower priority result when it is inserted.

Other configurable parameters of the Result Processor include RP_PROC_QUEUE_LENGTH and RP_QUEUE_LENGTH which control the number of processing requests the processing queue can hold and the number of messages the message queue can hold respectively.

The process flows for the Request Processor follow in Figure 36 and Figure 37.

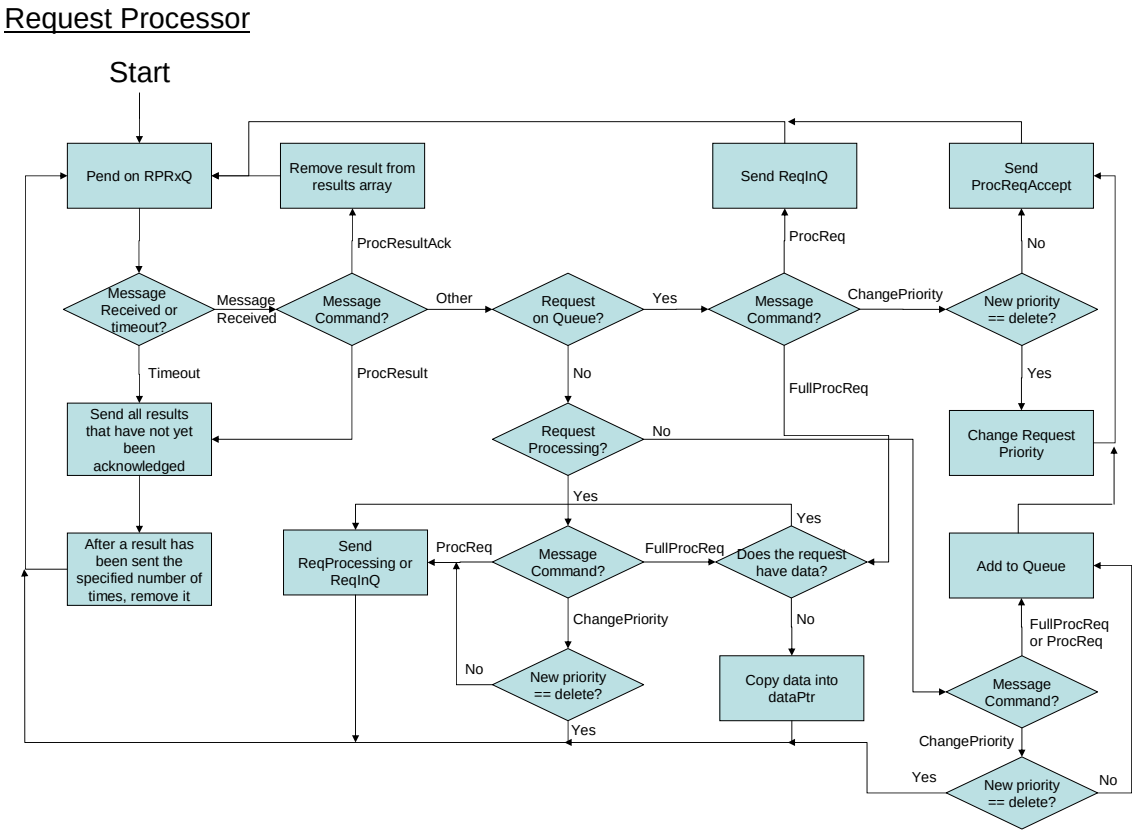


**Figure 36: Request processor process flow**

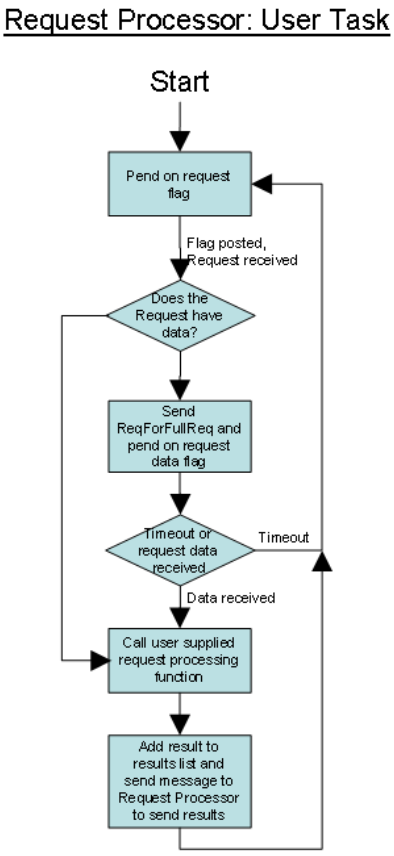


**Figure 37: Request processor user task process flow**

*App: Mode Changer*

The Mode Changer has the general task of managing the module to conserve as much power as possible. This is the only part of the application layer that will have to be specifically tailored depending on the attached resource (beyond simply changing a configuration variable). The Mode Changer has not been implemented in the first version of the MASS software, but in its initial design, it seeks to conserve power in four separate ways:

- Manages the power states of the attached resource
- Manages sampling rates when applicable
- Alters a modules actions based on the current configuration of the node
- Schedules module sleep times

The first duty of the Mode Changer is to manage the power states of the attached resource. Some resources, such as general purpose processors (GPPs), have several power states corresponding to varying levels of waking and varying amounts of

peripheral support turned on. The Mode Changer must thus understand the different states that the attached resource may be put into including knowing what the resources capabilities are in each state, the power drawn in each state, and the transition times between states. It may also completely power down the resource if the resource is only used when triggered by some other module.

In the case of sensors and wireless network connectors (WNCs), the Mode Changer manages sampling rates to conserve power. For a sensor, each sample taken generally represents a constant amount of energy expended, so adapting this sample rate to the expected number of events in the surrounding environment can help minimize power for a given application. In the case of WNCs, the transceiver may only be required to actively listen to a channel during certain periods of time and may be completely powered down otherwise. It is up to the Mode Changer to manage when and for what duration the transceivers will be actively listening ("sampling") to a channel.

The Mode Changer can also alter module actions based on the current node configuration. Since the net layer maintains knowledge about the other modules available in the system and also about the overall configuration of the node, the Mode Changer can use this information to control the resource more intelligently. For example, in a simple node where only a sensor and power supply are present, the Mode Changer on the sensor module may simply help store the collected data without trying to send it to a non-existent GPP or WNC.

Finally, the Mode Changer also schedules sleep times for its resource. In order to do this, it monitors the incoming request rate from other modules and uses this information to determine what the usage of the local resource is. If its attached resource is being used only infrequently, the Mode Changer will always put the resource into a low power state immediately after it finishes processing the requests on its Request Processor queue. If its attached resource is being used frequently, however, it may leave the resource running at its full power in expectation of receiving another request soon. This changing of sleep scheduling can therefore intelligently save the transition time and energy that would otherwise be required each time a resource is used, and adapt based on current usage conditions.

## Writing Code for MASS:

The MASS API consists of two main parts. The first is the specification of how to process incoming requests. The second allows the programmer (or user) to make outgoing requests. The result is a simple and robust system that relieves the user of worrying about how messages get from one module to another.

*Starting MASS:*

MASS is initialized by calling `InitMASS`.

- `void* InitMASS( INT8U type, INT8U subtype, void* (*RequestProcessingFunction) ( void*, INT16U, INT16U* ))`

The `type` argument is a consists of `MODTYPE_`[`GPP` | `SENSOR` | `WNC` | user defined] as found in global.h and must be specified. The `subtype` argument must be between 0 and 31 inclusive. The third argument is a pointer to a function which will be used by the request processor to handle requests received from other modules.

*Processing Requests:*

When MASS is initialized, `InitMASS` takes as input a function to process requests, `RequestProcessingFunction`. The signature of the function pointer consists of three arguments, a `void*` to the data to be processed, the length of the data to be processed, and a pointer to store the length of the result in. A `void*` to the result is returned. Request Processor calls this function on data received from other modules. The user must handle, process, or service the data in a way defined by the application. The user should not clean up any memory other than what it allocates. Specifically, the data to be processed should be treated as read-only and should not be freed.

*Sending Requests:*

Sending requests to another module is as simple as calling `Send`.

- `INT16U Send(INT8S prio, INT8U dest, INT16U dataLength, void* dataPtr)`

The prio argument specifies the priority of the message to be sent. The destination of the message is specified in dest, and can be either the default routing scheme, a broadcast, or another specific module in the node. The amount of data to send is indicated by dataLength, and the data itself is located at dataPtr. Once a message is sent, the user can check its status with two functions using the return value of Send.

- `MsgStatus GetStatus(INT16U key, INT16U timeout)`
- `INT8U GetRejections(INT16U key, INT16U timeout)`

GetStatus can be used to find out the status of the message, and GetRejections can be used to find out how many times a message has been bumped or been rejected. Both functions take as input the value returned by Send in key and a timeout in which to accomplish the task.

- `INT16U ChangeRequestPriority(INT16U key, INT8S newPrio, INT16U timeout)`

ChangeRequestPriority can be used to delete a message that is no longer necessary or change the priority of the message. In either case, the returned value of Send must be specified in key. The function also takes the new priority to assign to the message and a timeout in which to accomplish the change of priority.

*Configuring the Net Layer:*

The Net layer has several user-configurable parameters as well as an API function which allows the user to request information about modules currently active on the node. The maximum number of modules the Net layer can store information about can be modified by changing the value of MODULE_INFO_TABLE_SIZE. The number of messages that can be stored on the Tx and Rx queues is controlled by NET_QUEUE_LENGTH. Additional parameters control how module detection and network stabilization occurs include: HEARTBEAT_TIMER_MS, which determines how often each module broadcasts it's ID; HEARTBEAT_THRESHOLD, which determines how many heartbeat intervals a module must not be heard from before it is removed from

the list of active modules; NODE_TYPE_REDETERMINE_TIME, which determines the number of heartbeat intervals before the Net layer looks at the list of active modules and re-determines the type of the node.

The GetModulesByType function in the Net layer allows the user to request information about modules on the node based on their type. It takes 4 arguments: ( INT8U type, INT8U subtype, INT8U length, INT8U* err ). The type and subtype arguments are the same as those previously described, the length argument specifies how many bits from high bit to match on, where the type is in the high three bits and the subtype in the low five. The fourth argument is an error field, and the return value is a pointer to a linked-list of moduleTypeLLNode, whose structure is defined in global.h.

*Configuring the Transport Layer:*

The Transport layer has several user-configurable parameters which control how message fragmentation and reassembly works, as well as the usual parameter to modify the length of the Tx and Rx queues (TRANSPORT_QUEUE_LENGTH). The SHORT_MESSAGE parameter defines the longest processing request (including the message structure) that can be sent as a single stage request, as opposed to requiring a Processing Request followed by a Full Processing Request. The LONG_MESSAGE parameter defines the longest message that can be sent without fragmentation. The MESSAGE_FRAGMENT_LIFETIME_MS parameter determines how long the Transport layer will wait to receive all fragments of a message before discarding it. The FRAGMENT_CLEANUP_INTERVAL_MS parameter determines how often the Transport layer will traverse the fragmented message list and discard all fragments whose lifetime has expired.

*Configuring the Request Processor:*

The Request Processor has two user-configurable queue length parameters, RP_QUEUE_LENGTH controls the length of the incoming message queue, while RP_PROC_QUEUE_LENGTH controls the length of the processing queue. There are also several parameters which relate to how processing results are stored and sent. RESULT_ARRAY_ENTRIES determines how many processing results can be stored at a

time, when the results array fills up, the lowest priority results are bumped. `RESULT_SEND_ATTEMPTS` determines how many times results are sent without being acknowledged before they are discarded. `RESULT_RESEND_INTERVAL_MS` determines how long to wait between resending results. Finally, `FULL_REQ_TIMEOUT_MS` determines how long to wait for the data associated with a processing request before discarding it and moving on to the next request.

*Configuring the Local Event Handler:*

The user gives data to the Local Event Handler (LEH) by calling "`Send`." Send takes four arguments: `INT8U prio, INT8U dest, INT16U dataLength, void* dataPtr.` Prio is the desired priority of the request. `Dest` can be either `DEST_DEFAULT`, `DEST_BROADCAST`, or a specific module address retrieved from the Net layer. `DataLength` is the amount of data located at `dataPtr`. The user must then call `WaitForResult`. `WaitForResult` takes one parameter, `INT16U timeout`, which specifies how long to wait in clock ticks for a result. `WaitForResult` returns a pointer to a message.

The user can also inquire about the state of requests by calling "`GetStatus`" and "`GetRejections`" with the key returned by Send. The user can also delete a request or change a request's priority by calling "`ChangeRequestPriority`" with the key to the message and the desired new priority.

The LEH provides a number of parameters that can be configured to adjust its behavior. First, LEH_QUEUE_LENGTH is the number of entries the LEH's incoming queue is capable of holding. The maximum number of requests that the LEH can track is determined by LEH_TRACKER_ENTRIES. Most of the messages that the LEH sends generate responses. The total number of time to send a processing request before dropping the request is defined by PROC_RQST_RETRIES. Similarly, REJECTED_RETRIES defines how many times a request can be denied or bumped before it is dropped. Finally, FULL_RQST_RETRIES indicates how many times a full processing request is sent before it is dropped.

The time intervals associated with resending messages are also configurable. Each timeout is defined in milliseconds and converted to clock ticks at compile time. The

interval between sending processing requests is PROC_RQST_TIMEOUT_MS. The interval on which the LEH sends status requests to outstanding requests is defined by STATUS_TIMEOUT_MS. The number of milliseconds to wait between retrying requests that were rejected or bumped is REJECTED_TIMEOUT_MS, and the amount of time to wait for a full request accept before resending the full request is FULL_RQST_ACCPT_TIMEOUT_MS. In general, these timeouts should be on similar orders of magnitude, with STATUS_TIMEOUT_MS and REJECTED_TIMEOUT_MS larger than the other two.

## VISUALIZATION

In order to verify the functionality of the MASS system, an I2C sniffer was inserted into the communications traffic and a terminal program on a desktop computer could then monitor the bus. The traffic coming in from the bus was in a specialized format provided by the manufacturer of the I2C sniffer. The terminal buffer could be captured into a log file for storage and verification. A snippet of an example log file follows:

Start Time hh mm ss 18 47 21 Initial Status 0x81 User Terminated Monitor Initial Status 0x81 STOP

Sa00 Da2A Da00 Da00 Da00 Da02 Da00 Da02 Da69 Da50 Da13 STOP
Sa00 Da80 Da00 Da00 Da00 Da02 Da00 Da02 DaFA Da48 Da32 STOP
Sa00 Da6E Da00 Da00 Da00 Da02 Da00 Da02 Da86 Da40 DaA8 STOP
Sa00 Da6E Da00 Da00 Da00 Da02 Da00 Da02 Da4D Da40 Da63 STOP
Sa00 Da2A Da00 Da00 Da00 Da02 Da00 Da02 DaA5 Da50 DaDF STOP
Sa00 Da6E Da00 Da00 Da00 Da02 Da00 Da02 DaB3 Da40 Da9D STOP
Sa00 Da80 Da00 Da00 Da00 Da02 Da00 Da02 Da2A Da48 DaE2 STOP
…

**To From Flags Prio MsgID Cmd DataLength Data CheckSum Stop**

The first line of this log file displays initialization information as to the initial status and start time of the logging. Each of the following lines contains several bytes in hexadecimal notation prefixed by a two letter sequence “Sa” or “Da” and terminated with a “STOP”. The prefix “Sa” stands for start, the prefix “Da” stands for data, and the “STOP” stands for I2C stop condition. Thus there is always one start byte, followed by several data bytes, and terminated with a single stop condition. The ordering of these bytes is precisely the same as described above in the section “Messaging in MASS:”. In other words, the first byte is the destination address (“00” is the broadcast address), the second byte is the source address, the third byte is a flags field concerning transport layer fragmentation (“00” indicates no necessary transport layer action), the fourth byte is the priority of the message (“00” being the highest priority), the fifth byte is the message ID number (which in the case of these messages does not matter), the sixth byte is the command (“02” is an IDBroadcast command), the seventh and eighth bytes are the number of appended data bytes (in this case “0002” → 2), the next DataLength bytes are appended data, and the last byte before the stop condition is a checksum. The interpretation of these log files can thus provide an insight into the operations of a MASS

stack for any given traffic on the bus, and they were used extensively for testing and debugging purposes.

While these log files are readable to the trained eye, they are quite cryptic to a MASS beginner. In order to alleviate this difficulty, a visualization of these log files was thus created. The visualization simply parses the log files line by line and animates the messaging actions. Figure 38 below shows two examples of visualizations of the same log file at different points in time. (Note: The log file used for these examples is the same as above.)

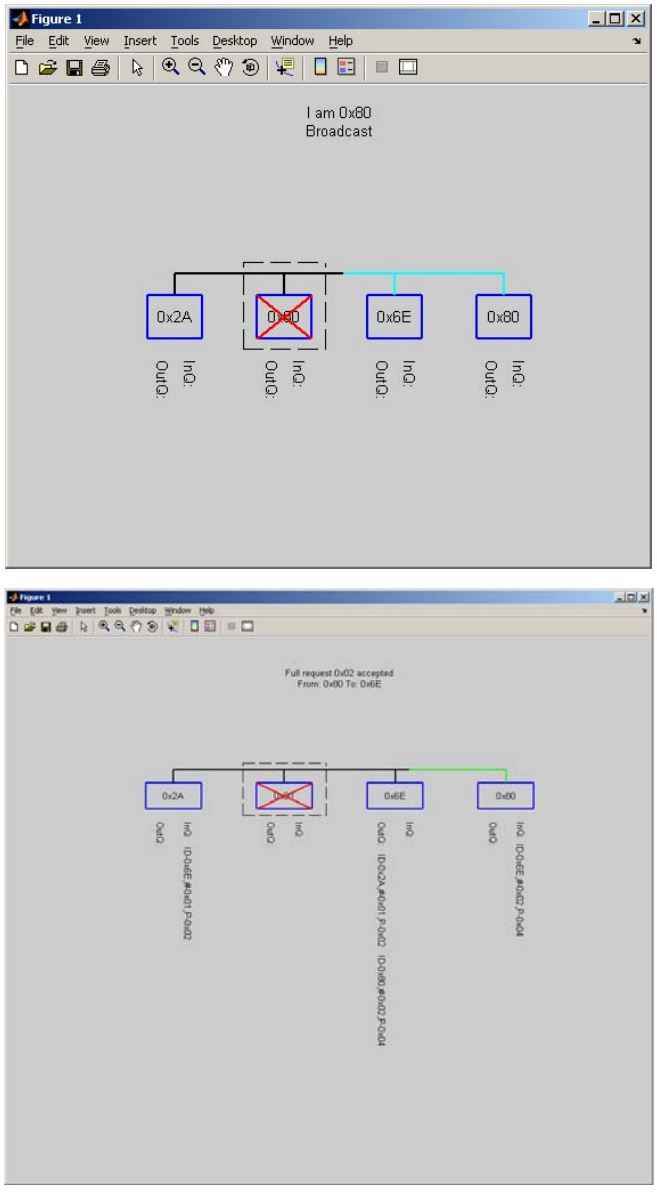


**Figure 38: Visualization examples**

The visualization uses color coded animation to cue the observer as to the actions occurring on the bus. Blue indicates requests or notifications, green indicates positive responses, yellow indicates warning responses, and red indicates negative responses. Most importantly, the visualization also displays a translation of each line in the log file into English at the top of the screen. These translations are the critical component of the visualization and allow an unfamiliar user to quickly determine current node state and the meaning of the current message on the bus. The visualization also keeps an InQ and OutQ for each module keeping track of the requests currently in the Local Event Handler and Request Processor. The messages on the queues are identified by who they are from or to as well as their message ID and priority. Finally, the visualization keeps track of which modules are active and which have become de-active in the system. As defined in the Net layer, if “heartbeat” messages are not heard by the other modules at some minimum rate, a module will be dropped from the module list of the other modules in the system. The visualization thus keeps track of the “heartbeat” messages sent by each module and if the appropriate number of messages are not sent, it is Xed out of the system.

## CONCLUSION

A modular architecture for sensor network nodes was described. The need and justification for the development of the novel architecture was explained, and a high level description of the proposed architecture was presented. The architecture was then analyzed mathematically with respect to a previous standard centralized architecture and numerous control mechanisms and tradeoffs were determined. The full process of implementation was also discussed from a high level view of the software architecture through topical simulation of the design and finally through each detailed layer of the software. The software was designed to have a simple user API and to be flexible, extensible, and modular for ease of use and maintenance. Future intended developments of the Finally, a visualization tool used to debug and monitor system state was also described.

This architecture has been submitted for two United States patents – one for the hardware design and one for the software design. Young versions of the architecture have already been used in two prototype applications for rapid response and perimeter security type missions. The architecture will continue to be developed and used, and comparisons to traditional architectures will be analyzed experimentally. As the architecture is integrated into more applications, the proposed power savings and performance benefits as well as programmatic time and money savings will be determined critically.

## ACKNOWLEDGEMENTS

We would like to first thank the Sandia National Laboratories Computer Science Research Foundation for sponsoring this work. Nothing could have happened without its support. We would also like to thank the Embedded Reasoning Institute for leveraging some of its own resources in order to help support and lend guidance during the development of this architecture.

# APPENDICES

## *Appendix A: Examples of MASS functionality*

| Module 1 | Module 2 |
|---|---|
| 1. User requests processing task<br>2. Local Event Handler generates a processing request, adds the request to the tracker, and passes it to the Transport layer<br>3. Transport layer fragments message if necessary and passes it to the Net layer<br>4. The Net layer addresses the message using the default routing scheme if the message is not already addressed | |
| | 5. Message is received by the Phy layer, assembled by the Transport layer, and posted to the request processor via the RPRxQ<br>6. Request Processor posts the message to the processing queue |
| 7. Tracker requests status of request | |
| | 8. Request processor sends request in queue message<br>9. Request Processor sends the request to the User layer request processing task<br>10. The User layer request processing task processes the request, adds the result to the results list, and notifies the Request Processor<br>11. The Request Processor sends the result of the processing task to the module which requested it |
| 12. The local event handler acknowledges the receipt of the processing result | |

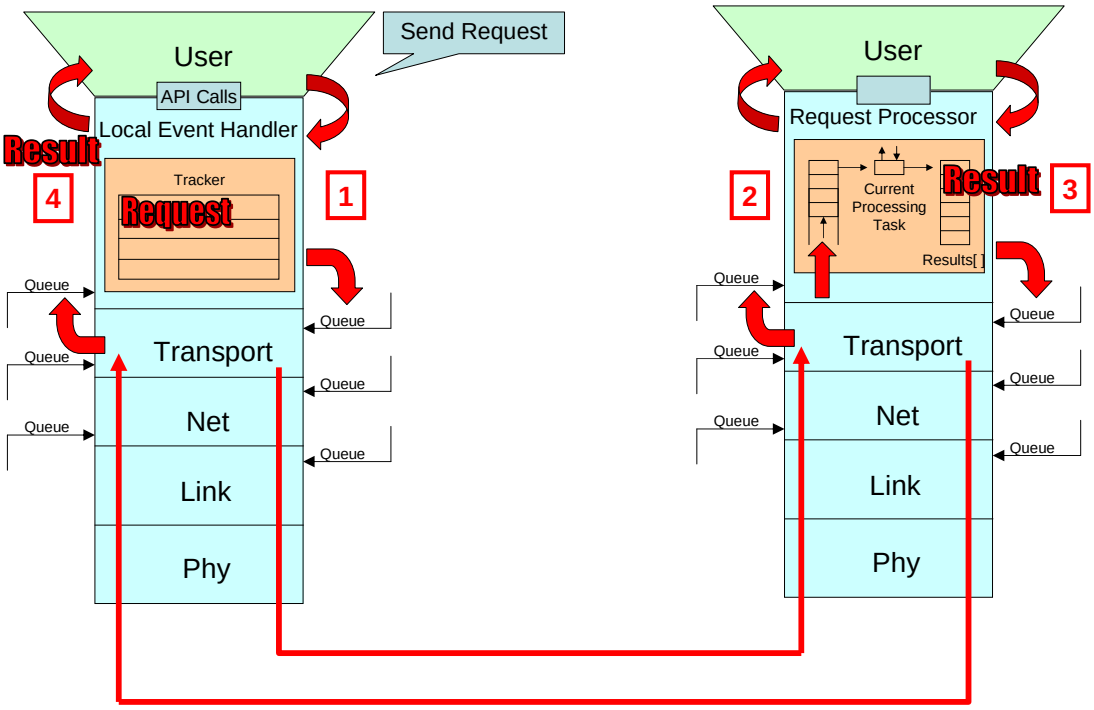


**Figure 39: Example of MASS functionality**

## DISTRIBUTION

| | | | |
|---|---|---|---|
| 1 | MS9915 | Jesse Davis | 8961 |
| 1 | MS9154 | Doug Stark | 8245 |
| 1 | MS9101 | Ron Kyker | 8245 |
| 1 | MS9101 | Chris Kershaw | 8232 |
| 1 | MS9915 | Nina Berry | 8961 |
| 1 | MS9158 | Teresa Ko | 8961 |
| 1 | MS9158 | Rob Armstrong | 8961 |
| 1 | MS9158 | Mitch Sukalski | 8961 |
| 1 | MS9401 | Greg Cardinale | 8245 |
| 1 | MS9151 | Jim Handrock | 8960 |
| 1 | MS9153 | Brian Damkroger | 8240 |
| 1 | MS9003 | Ken Washington | 8900 |
| 1 | MS9153 | Doug Henson | 8200 |
| 1 | MS9018 | Central Technical File | 8945-1 |
| 2 | MS0899 | Technical Library | 9616 |
| 1 | MS0612 | Review & Approval Desk for DOE/OSTI via URL | 9612 |
| 1 | Nick Edmonds<br>2200 W Sudbury Dr.<br>Apt. B-08<br>Bloomington, IN 47403 | | 8961 |