Dynamic Management of Cooperating Peers in Wireless Embedded Systems: An Approach Driven by Information Quality

David A. Robinson and Karsten Schwan

Georgia Institute of Technology, Atlanta GA 30332, USA
robinson,schwan@cc.gatech.edu

Abstract. Data-intensive applications like remote sensing or peer-to-peer communications are increasingly prevalent on wireless-connected, pervasive systems platforms. Since resource limitations make it infeasible to deliver all data to all participants at all times, these applications' data streams must be managed at runtime. Our approach is to continuously manage data streams in terms of the quality of the information received by end users. Methods are developed that dynamically optimize application-relevant metrics of *information quality*. These methods are realized in data delivery middleware where information quality is managed via dynamically deployed and adapted application-specific, source-resident data filters. Experiments are performed with portable devices that operate across 802.11b-based wireless networks, using camera-captured images as sample sensor data and using filters that implement a variety of data downsampling methods, with consequent effects on information quality. Measurements show that (1) dynamic management of information quality is feasible and generates quality gains compared to non-managed approaches, and (2) that the complexity of multi-sensor, multi-client quality management indicates the use of heuristic solution algorithms specific to certain system configurations.

1 Dynamically Managing Information Quality

Dynamic pervasive systems and applications. Pervasive systems are becoming increasingly complex, ranging from a single sophisticated handheld PDA [2], to multiple PDAs/sensors operating jointly when providing information to collaborating end users [37], to entire distributed systems in which 'field' personnel use portable devices to wirelessly interact with distributed sensors and to communicate with each other and with central sites [14, 26], via heterogeneous network infrastructures [10]. Such complex systems' high levels of dynamics and distribution imply that individual devices and/or subsystems come and go frequently, that connectivities vary, and that end user needs and behaviors change frequently. Yet end users desire suitable service whether operating autonomously, ill-connected, or well-connected. One concrete metric is timely information delivery, even in the presence of connectivity and power constraints. Stated more

generally, end users require certain levels of quality from the information delivered to them, where *information quality* is a multi-dimensional metric composed of application-specific measures of quality (e.g., data resolution) and of system-centric measures of information delivery (e.g., end-to-end delay).

This paper explores the dynamic management of information quality for distributed embedded applications. One application is a large number of clients accessing dynamically created information via shared resources, as exemplified by fans in a stadium watching selected details of a football game that is captured by multiple cameras distributed across the field, as shown in Figure 5.1[1]. Distributed sensors (e.g., cameras) share the bandwidth available via the base stations being used, as do multiple clients, but clients don't share bandwidth with sensors. Since each particular image stream (from a certain sensor, or to a certain client) is important and since bandwidth is shared, it is simply not feasible to broadcast all sensors' data to all clients at all times. These constraints, therefore, create interference across multiple data streams, because some sensor data stream that is transmitted reliably can consume sufficient resources to consistently slow down other streams. Slowdown may be due to the shared bandwidth consumed by multiple streams, a fact that is addressed by military network standards like slotted ring architectures[15], for example. More subtly and not addressed by network architectures, but important at the middleware level [11, 38], is the fact that a single slow consumer of reliably transmitted, time-sensitive data[2] may slow down data delivery to other consumers due to buffer limitations at data sources. Furthermore, even with unlimited buffer sizes, the data delivered may become 'stale', due to backlogs of older data. We call this problem *data stream interference.*

In the stadium example described above, as well as in more complex pervasive applications [10], a concrete way to manage information quality is to perform data downsampling or compression at data sources and at intermediate data brokers (if necessary). Such management has to be dynamic, not only because resources vary but also because the end user should get the information that best serves their current needs, the latter being subject to change over time. A simple instance of dynamic management studied in this paper is when a single source provides data to multiple clients, where clients' data quality is dynamically managed via downsampling, thereby maintaining suitable data rates to multiple clients despite radpidly changing network conditions.

Our quality-centric solution approach to dealing with data stream interference differs from previous work (e.g., multimedia servers[32]), which has typically devised explicit methods for synchronizing, throttling, or otherwise controlling the production rates of data viz a viz clients' consumption rates, sometimes including complex methods for distributed bandwidth and resource scheduling[23].

[1] A UAV scenario is another example[14, 26].

[2] Sensor data fusion imposes even more stringent requirements, like requiring time-synchronized data across multiple streams.

By managing the *quality of information*, we implicitly and continuously adapt the resource consumption of the information streams produced, manipulated, and transmitted in distributed embedded applications. Resource consumption is adapted by changing the processing actions that create meaningful information from the raw data captured by sensors, thereby affecting the bandwidths and CPU cycles consumed by these flows. The adaptations performed are application-specific, thereby directly addressing a system's ability to meet current end user requirements. Requirements of end users are stated in terms relevant to them and mapped to the aforementioned information quality metric.

To better understand the dynamic management of information quality, consider an emergency response scenario, where rescue workers use PDAs/cellphones equipped with cameras and microphones to coordinate their actions. When a worker produces an important video feed (e.g., showing a 'safe' passage in and out of the emergency site), this stream's importance may be mapped to high information quality compared to other streams, thereby prompting middleware to focus on this vs. other information streams, transmitting it reliably, and directly to whoever needs it, such as other workers, rescue vehicles, and command centers. The need for reliability links a source's production rate to sinks' rates of consumption, so that a slow subscriber to such information can slow down all others. Slow information consumption can be due to many factors, including end user mobility or their current PDA usage. This problem also extends to totally ordered reliable multicast, where a sender will eventually be slowed down by a slow client, unless NAKs are used in place of ACKs and information is only buffered for certain lengths of time [13]. Conversely, when information from multiple sensors must be fused, as with sensor data processed at central headquarters or at a larger emergency vehicle, a single slow sink (or source) has similarly global effects. Finally, it is clear that each node has individual preferences as to what information is relevant (or not) to its current mission. Therefore, information quality metrics cannot be based on syntactic measures, but instead, must take into account semantic information derived from specific clients' interests.

Implementation and Results. Our solution approach is implemented in the context of the ECho publish/subscribe infrastructure, where any end user can dynamically subscribe to any information flow of current interest. ECho [11] implements event channels to which information providers (publishers or sources) and information consumers (clients or sinks) subscribe whenever interested. Information is published on a channel when it becomes available and transported immediately. Information transmission is not performed via intermediate brokers, as in other publish/subscribe systems [5, 8]. Instead, event channel subscribers are linked via point-to-point, reliable network connections. Further, information published on channels is transmitted only when there are current (i.e., interested) subscribers. Finally, information submission or receipt is in terms of well-defined, application-level typed information units, termed *events*, and both such actions can be linked with the execution of *handlers* on events. Such handlers are the entities used and adapted to manage the quality of information in event streams.

The main results attained in this paper are as follows. For a single sensor and multiple clients or for multiple sensors and a single client, respectively, we demonstrate scalable (in terms of numbers of clients or sensors) and efficient algorithms for dealing with data stream interference, based on manipulating the quality of the information produced, transmitted, and received. These solutions utilize event handlers at data sources that are dynamically adapted in terms of their event processing and, therefore, of consumed bandwidth and event sizes delivered to sinks. Improvements are demonstrated experimentally for multiple devices wirelessly communicating via a single base station. When using rate as a metric, such improvements can more than double the rate of streams to slow clients, thereby substantially improving the average data transfer rate experienced by all stream clients.

Overview. In the remainder of this paper, an informal description of the stream interference problem is followed by its precise mathematical formulation. Based on this formulation, a solution heuristic is developed and then deployed in prototype middleware for data distribution in pervasive applications. Experimental evaluations utilize both in-lab and campus-wide wireless network infrastructures.

2 Related Work

Previous work on stream scheduling and synchronization typically relies on explicit solutions, such as those developed in distributed real-time scheduling [23] or in multimedia services [32]. In comparison, our work uses an implicit, adaptive approach to distributed system management [6]. Unlike research in adaptive multimedia [18], we do not permit data loss, in part due to the applications we target (e.g., remote sensing). Nonetheless, the packet scheduling algorithms [36] or the control theoretic models developed in the multimedia domain [22, 1, 24] could be applied to further improve the basic results presented in this paper. Specifically, such results could help refine the actual adaptations applied to information streams. In addition, concerning Quality of Service allocations to many clients, dynamic estimate refinement could be used to automatically determine the current quality of information desired by end users, including based on past user behavior [7]. Finally, previous work has developed general approaches for mapping user-level quality of service specifications onto policies that scale resource consumption, using utility/payoff functions and priority packet dropping for MPEG streams, for example [20, 21]. In comparison, we use simple, intuitive mappings, without claiming their general utility or validity.

Our optimization heuristics can be generalized to consider multiple types of resources, by using more general frameworks for adaptation like those presented in [31, 17] or in [35]. Further improvements would result from their support by middleware or by operating system constructs [27, 29]. Future work should consider issues beyond the CPU and communication constraints raised in this paper, such as the power requirements of mobile and wireless devices [12, 21, 30]. Finally, a limitation of our current approach is that information streams are

manipulated only at sources or sinks, not at intermediate sites. In overlay networks [4], intermediate brokers can run computationally intensive filters and/or deal with network heterogeneity [9], thereby permitting solution configurations not yet considered in our work [33].

3 Problem Description and Solution

3.1 Informal Problem Statement

Information quality is a multi-dimensional metric that captures both platform- and application-centric measures about information streams. In remote sensing, information quality may be described as a floating point number derived from the attained frame rate and end-to-end delay, from each frame's width, height, number of color bits, and from the client's *synchronization window* with respect to other clients subscribing to the same data stream. The synchronization window uses frame ids to express how 'in sync' two clients are with respect to the information units they receive. Specifically, two or more cooperating clients are 'window synchronized', if for each frame arriving at a client, its sequence number is contained in all of the clients' windows. When this it not the case, a synchronization exception is raised. In a sense, this window captures the 'freshness'[19] of information used by one client compares to that used by another. In the remainder of this paper, we use window synchronization as a concrete contribution to the measure of information quality.

A simple problem expressed with synchronization windows is one in which a single source sends data to multiple sinks. When one sink falls behind, this 'straggler' client can cause high buffer fill levels at the sender and can thereby slow down other receivers. A solution based on information quality is one that downsamples the events sent to the 'straggler'. This reduces both the bandwidth required for transmitting events and the CPU cycles used by the client for event receipt and display. Note that the inverse problem may be formulated in a similar fashion. Here, multiple sensors produce qualitatively different data, and receivers must compose or fuse time-near data items. Synchronization windows for clients express their tolerances for 'out of sync' data. Tolerances may stem from information 'freshness' requirements or from client-side buffer limitations. Information quality measures use synchronization windows as well as application characteristics like image resolution, since sensor fusion with higher resolution images typically produces superior results.

3.2 Precise Formulation

Consider the single source sending data to multiple – m – sinks. The following quality function $qu_i(wi, hi, fr, la, cb, cp)$ may be defined, varying across sinks and returning values in the range of $0.0 - 1.0$. Here, wi is the width of the sensor image frame produced and transmitted by the source; hi is the height; fr is the frame rate in frames per second; la is the latency or end-to-end delay of frame transmission; cb is the number of color bits per pixel; and cp is an integer denoting the use of either lossless, lossy, or no compression. These are simply 6 different dimensions of quality suitable for video streaming applications, as shown in [25].

Let c_i be a 6-tuple containing the current values of the quality function's parameters for sink i. The sensor-side processing and transmission costs associated with these quality parameters are mapped to utilization values ranging from 0.0 to 1.0, termed $cpu_i(c_i)$ (processing costs) and $bw_i(c_i)$ (bandwidth usage), respectively. For the sensor CPU, there are are also scb_t (sensor CPU background traffic) individual portions that are unavailable for sensor processing actions. On the receiver-side, the set of $c_i, 1 \leq i \leq m$ also consumes some amount of CPU time, resulting in utilization values ranging from 0.0 to 1.0, termed $rcv(c_i)$ (receiver cost). For each of the m receivers, the unavailable portion is denoted by $rcb_{i,t}$ (receiver CPU background). Finally, let bbw_t (background bandwidth) be the amount of bandwidth that is unavailable at any given point in time t, typically due to cross traffic.

Given these formulations, solution constraints are:

- Sensor CPU limitation
  $\sum_{i=1}^{m} pc_i(c_i) + scb_t \leq 1.0$
- Receiver CPU limitation
  $\forall i_{1 \leq i \leq m} sc(c_i) + rcb_{i,t} \leq 1.0$
- Total Bandwidth limitation
  $\sum_{i=1}^{m} bc_i(c_i) + bbt_t \leq 1.0$

The objective function for information quality-conscious sensor rate optimization, then, may be formulated as:

- For each time step t, maximize $s = (min_{1 \leq i \leq m}(qu_i(c_i)))$.

Optimization fails, that is, constraints and starting values are infeasible, when $s = 0.0$. Note that this formulation does not explicitly consider the timeliness of the input values used, implying that system monitoring is precise and instantaneous. Further, realistically, complete knowledge of functions scb, rcb, and bbt does not typically exist, implying that the functions used in the model only approximate reality. Mote that this precise problem formulation is easily changed to one in which multiple sensors provide data to one sink.

Consider a sensor producing images used by two different clients. Two dimensions of information quality are used, such as data rate and image resolution. In this case, the clients' data rates may be synchronized by manipulating these quality dimensions using the following heuristic solution algorithm. The algorithm has two phases: (1) constraint satisfaction, followed by (2) optimization.

The first phase of the heuristic attempts to meet application constraints. For each client, binary search over the possible quality values, only considering changes to one dimension at a time. Consider changes first down and then up. Proceed until each client has attained some positive total quality.

Make the image resolution and data rate high enough that positive quality is achieved for each of the clients. Let's say the data rate is 100 kilobytes per second

and the image resolution is 50 thousand pixels, just for the sake of argument. In our example, a solution to the first phase may not be feasible with respect to available resources. Let's suppose that this is not the case.

For the second phase, all of the CPU and network resources are divided and credited to the sensors. Then, a 'downward' binary search, again for each dimension individually, attempts to meet the resource constraints, for each resource needed. Total quality must be positive. If it is not, then that 'possibility' is eliminated. This approach causes clients to remain within stated quality constraints. The other constraints are ordered. They can be ordered any way one wants, and one meets constraints earlier in the ordering and 'holds on' to them in later repetitions. In this way, the algorithm seeks a solution. After a few repetitions, which is a predefined number, the search stops, and every sensor that does not use all of its credited resources donates them to sensors that need more CPU and network. Then, the algorithm runs again, just the way it did before with no modifications except for the resource constraints, for the sensors with insufficient resources, for the same number of repetitions. Finally, if each sensor's resources are not exceeded and all of its constraints are met, then this phase of the algorithm succeeds. Otherwise, it fails.

The second phase is only carried out for sensors that do not meet all of the constraints in the first phase. In our simple example, let's say that the algorithm succeeds, but the data rate for the clients is half of that quoted above, due to needing to meet the constraints imposed by the limited bandwidth and CPU in the network. The complexity of the constraint matching algorithm is $O(n \lg m)$, with n representing the number of sensors and m representing the maximum number of quality levels in any of the quality dimensions being used.

The optimization phase is similar to constraint satisfaction. First, if the constraint phase fails, then optimization fails, as well. Otherwise, optimization proceeds using a hill-climbing approach. The idea is to first reduce information quality for the client with the highest total quality value. Again, binary search is used along each quality dimension, where the client with the lowest ratio of reduced quality to resource reduction is chosen. In this way, the algorithm employs a greedy hueristic. The quality dimension that is increased in turn is the one with the highest ratio of improvement in quality to resources used, chosen from among all quality dimensions and resources. Resources are weighed according to their scarcity. Hence, whatever resources are causing the the bottleneck in the system 'cost' more. Ties are broken randomly. In our example, CPU resources are plentiful, but network resources are scarce. Thus, the algorithm continues, causing heavy compression because that method can raise image resolution while using the same data rate as before. So, let's say that the client data rates are 55 kilobytes a second, but the image resolution is 150% as much as it was before. This procedure is repeated until it fails to produce further changes.

The complexity of this heuristic is $O(max(m, n) * \lg m * dim)$ time, where dim is the number of dimensions of information quality, and m is the maxi-

mum number of levels in any of the quality dimensions being used. The term $max(m, n)$ appears because the algorithm must determine whether there are multiple minima. The complete details of the algorithm ,and it's big "0" analysis, appear in Appendix A, at the end of this paper. Note that the sensor data rate synchronization uses the same algorithm as the one discussed above, except that the roles of clients and sensors are reversed.

While seemingly complex, for systems of reasonable size (in terms of numbers of clients) and complexity (in terms of quality dimensions and levels), our heuristic attains solutions quickly. To demonstrate, we run the heuristic for representative cases with 10, 50, and 100 clients, respectively, assuming the aforementioned 6 quality dimensions, x levels of quality per dimension, and the resources available on a typical laptop (the MPIII described in detail in Section 5.1). With results averaged over three runs, both the optimization and the constraint algorithms are shown to be roughly linear in the number of clients, and more importantly, the algorithms are shown to run faster than 10 milliseconds even for 100 clients. This indicates that the dynamic management of information quality is a practical approach to managing distributed processes in embedded systems.

	constraint	optimization
10 clients	.423 ms	1.064 ms
50 clients	2.098 ms	4.946 ms
100 clients	4.156 ms	9.694 ms

4 Prototype Implementation

The application is controlled with control messages being sent back from each client to the server that is sending it data sometimes when a frame of video is received or when requirements change, or even after a certain number of seconds have elapsed. The control message contains the client's desires about what video format and what synchronization it would like, plus what frame number or timestamp the client received, plus some network-level information from the client to the sensor, that mainly describes network conditions and the CPU usage. This is basically a control loop between the client and the sensor. The server sends video, the specific format of this video frame, and some network information to the client. The sensors also send some control information among themselves in order to set up the algorithm, but only in cases where there is more than one information source. A machine may be a client, a sensor, or both.

5 Implementation and Evaluation

5.1 Implementation

Figure 5.1 depicts the ECho-based realization of an adaptive application with continuous quality management. Primary data channels are shown with bold lines, whereas control channels are depicted with dashed lines. The control information exchanged between receivers and sensors concerns setup and teardown

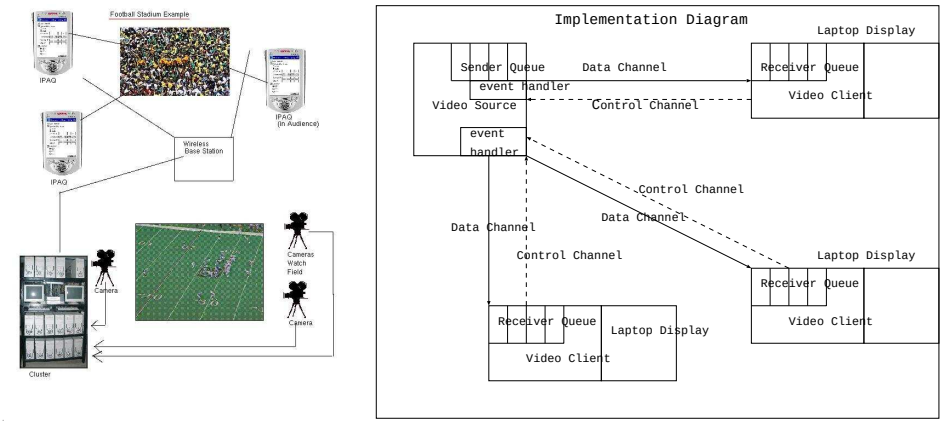

Fig. 1. Left: Data Capture and Distriubution in an Equipped Stadium, Right: Software Architecture of Implementation

and more importantly, includes notifications of changes in requirements, indicating a receiver's desires in terms of data format, permissable synchronization window, and updates on current, observed network state and CPU load. Monitoring information is typically piggybacked onto the sensor data exchanges ongoing in the system, but occasionally requires additional control exchanges.

Network measurements are provided by the communication layer, in our case using an instrumented version of reliable UDP (RUDP), a protocol used with the ECho middleware and developed at Georgia Tech [16]. Measurements are exchanged between machines as summary information several times a second. Given these values, the wireless network's state is modeled by a finite state machine, with each state describing a certain level of loss rate and load. Also taken into account is whether the wireless network is available, temporarily disconnected, or long-term disconnected. CPU measurements are performed locally and infrequently exchanged across sensors and receivers. As with the network, each CPU is modeled as an automaton with states corresponding to some small number of load levels, expressing very low, acceptable, and high loads.

Data downsampling filters are conceptualized as functions that take in data, manipulate it, and then output some data. All such data is described by well-defined data event formats. Filters may reside at any physical processor, and their computational and data copying functionality defines their costs, measured as their execution times.

For brevity, we do not describe the implementation of the continuous monitoring and adaptations performed in the system. The reader may refer to [31]

for a more general implementation of a distributed adaptive system, including a detailed discussion of adaptation overheads and delays.

6 Experimental Evaluation

6.1 Testbed Description

The sample portable devices used in our experimental evaluation are three laptops running Linux, but equipped with CPUs of different speeds and with different memory and graphics capabilities, the intent being to model future heterogeneous handheld devices. The first device is a Mobile Pentium III with a 30 gigabyte hard drive, 256 megabytes of system memory, a 1.13 GHz CPU, and an Nvidia GeForce 2GO video card with 16 megabytes of memory. The second machine is a 400 MHz Pentium II with a 12 gigabyte hard drive, 128 megabytes of memory, and a NM2360B graphics chipset with 4 megabytes of video memory. The third is a 700 MHz Dell Latitude L400 Pentium III with 128 megabytes of RAM, a 10 gigabyte hard drive, and an AGP-ATI Rage Mobility graphics chipset. All devices communicate via the same 802.11b Lucent Orinoco cards, using a single basestation not shared with any other devices.

Note that compared to the laptop devices used here, the current generation Compaq iPAQ Pocket PCs already run at 400 MHz and have 64 megabytes of memory. Their next generation is likely to approach the power of the laptops used in our evaluations.

6.2 Microbenchmarks

Microbenchmarks diagnose the basic ability of portable devices to run remote sensing applications and specifically, handlers that manipulate image content prior to transmission. A second intent of microbenchmarks is to establish the tradeoffs in terms of CPU usage and communication with respect to the total end-to-end delay experienced by an application. Tradeoffs exist because a slow machine may have more difficulties compressing data to reduce transmission delay (and the bandwidth used) than a fast machine, for instance.

First and foremost, simple image downsampling like grayscaling color images or image cropping experiences very low delays. Rather than reporting such numbers, for more complex methods like image compression, computation times scale with image sizes, and so do the times required to display an image, as evident from the measurements in the table below. Differences in processor speeds and in the capabilities of graphics chipsets have the obvious effects. The table lists the times (in seconds) per frame required for sts the times (in seconds) per frame required for JPEG compression (JPEGC) and JPEG decompression (JPEGD), and for Lempel-Ziv compression and decompression. The image sizes used are 160x160, 320x240, and 640x480 for small, medium, and large images, respectively. Display time is the total time required to display an image after it has been received. The measurements reported are in seconds per frame, computed as averages of three runs over 1,500 image frames (1,000 frames for the display measurements).

	JPEGC	JPEGD	LZOC	LZOD	DISP
MPIII					
Small	.004862	.007459	.003243	.00233	.002235
Med	.02013	.01807	.01718	.00261	.007518
Large	.1163	.1067	.03293	.0306	.03548
PII					
Small	.01987	.01526	.1422	.003784	.004523
Med	.06126	.05918	.03987	.01194	.01356
Large	.2805	.1891	.1044	.04354	.1309
L400					
Small	.007509	.005632	.005045	.001279	.004613
Med	.02988	.02386	.03048	.006290	.01626
Large	.1458	.09536	.66146	.02021	.07863

The next two tables depict the costs experienced by an alternative approachto image downsampling, where images are converted into tracks, and track lists (not images) are transmitted from sensors to clients. While the simple tracking algorithm (blob tracking) has performance comparable to compression, a more realistic algorithm (using feature detection) has substantially higher costs (see [34] for details about the image tracking algorithms used in our work). An additional machine, a dual processor Pentium II with 512 megs of RAM, is shown as another useful data point. speed?? Simple Image Tracking (in secs.):

Size	PIII	L400	PII	DPII
Small	.001782	.002847	.005889	.006508
Med	.006640	.01356	.03202	.02531
Large	.07616	.1287	.2437	.20998

Complex Image Tracking (in secs.):

Size	PIII	L400	PII	Dual PII
$160x160$	.06773	.1089	.2191	.1833
$320x240$	.2572	.4307	.8723	.6939
$640x480$	1.230	2.111	4.092	3.340

Network measurements show that image transmission costs (delay per frame) approximate the costs of complex data handlers like image tracking. In contrast, simple downsampling methods like grayscaling have overheads that are easily amortized by the consequent reductions in image transmission costs. Further, even for moderately complex operations on images (including compression), it can be advantageous to first compress and then transmit data, as evident from the table below in which uncompressed 24 bit image frames are transmitted across 802.11b links. Measurements are averaged over 3 runs, each run using 1,000 frames.

$160x160$	.1260 seconds/frame
$320x240$	.3717 seconds/frame
$640x480$	1.472 seconds/frame

While frame transfer delays can be substantial, the end-to-end latency experienced by small control messages is acceptable. For instance, considera configuration in which the PIII acts as a sensor device and the L400 actsas a client. A control message from the L400 to the PIII to change image characteristics (e.g., image size) takes effect in only a few seconds, with delay average and variance increasing for larger number of clients (e.g., for two clients versus a single client). Increases reflect the additional control messages being sent and the fact that in

these measurements, control channels share bandwidth with the ongoing data transmissions.

Control Message Latencies (in secs.):

Num Clients	max	min	average
1	2.380	.9187	1.483
2	3.490	.9598	1.885

The final set of microbenchmarks compares the ideal laboratory measurements reported above with measurements of actual delays experienced in a busy campus-wide wireless network. Here, two devices communicate via the LAWN Georgia Tech campus wireless network, using two different sets of locations. The CCB set refers to a configuration in which one machine in the first floor of the College of Computing (CCB) building communicates with another machine in the third floor. The Library set has the third floor CCB machine talking to another one in the campus library, which is about 1/2 mile away from the CCB building. The measurements are taken at about 9:30 A.M. - 10:00 A.M. on a Monday, andthe again at about 4:30 P.M. - 5:00 P.M. on another Monday. The values reported are average numbers aggregated over three runs of 1,000 latency measurements each.

Control message latencies in a campus wireless network (in secs.):

Place	Minimum	Maximum	Average
CCB	.006008	.025269	.006761
Library	.000894	.044862	.007160

Measurements demonstrate the substantial effects of multi-hop vs. single-hop communications, with delays ranging from approx. 9 to 45 milliseconds, and with larger delay variances experienced by multi-hop communications.

6.3 Performance Evaluation

Microbenchmarks demonstrate that source-based filtering can improve the end-to-end delays experienced from the time an image is captured by a sensor to the time it is displayed by a client, with consequent reductions in information quality. We next demonstrate that even our relatively simple optimization algorithm can realize improvements in the information quality experienced by end users, under conditions in which resource availabilities vary dynamically. In the measurements below, except for the 'straggler client' experiment, we attempt to synchronize client data rates for 1 sensor and 2 clients, using the laboratory configuration employed in the microbenchmarks.

The experiment shown in Figure 6.3 (left side) simply demonstrates that a single 'straggler' can indeed slow down another sensor data recipient. In this experiment, one of two clients of the same sensor data stream (using 160x160 size image frames) is slowed by a CPU perturber (see time 60 in the graph), and this slowdown soon results (see time 70) in a consequent slowdown of the other client. The perturbation is removed at time 180. The model presented in Section

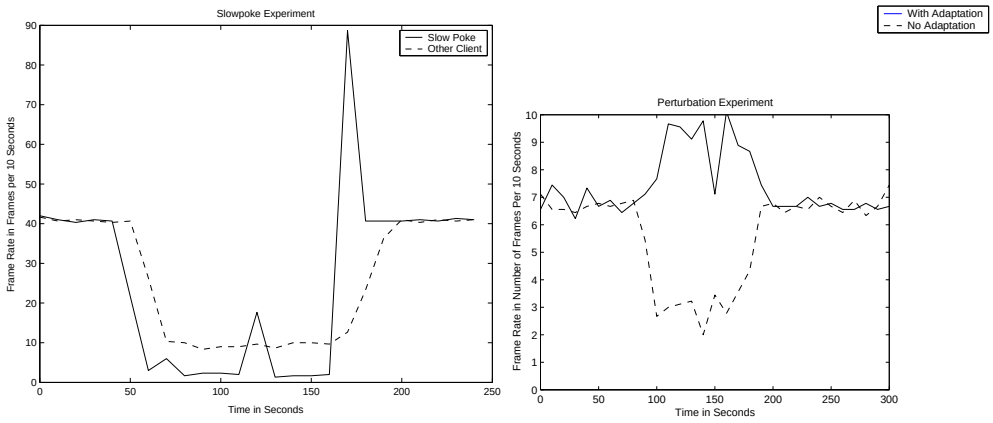

Fig. 2. Left: Solid Line: The Straggler. Dashed Line Affected Client., Right: PII's CPU Perturbation. Solid Line Adaptive Version. Dashed Line: Non-Adaptive Version.

3.2 represents this situation as one in which the straggler's scb_t values increase, followed by a decrease when the perturbation is removed.

6.4 Utility of Source-based Downsampling

Figure 6.3 depicts measurements that underline the validity of our approach. Here, the information stream received by a PII-based client is downsampled at the source, in response to that client's perturbation by a process that consumes almost 90% of its CPU. Two scenarios are shown. In the one indicated by the solid line, at time 90, the PII is not only perturbed, but in addition, the PII-directed stream is downsampled using the image tracking method explained in Section 6.2. This removes sufficient load from the PII to prevent it from experiencing the perturbation and consequently, preventing slowdown for other clients. In fact, in this case, the PII runs a bit faster because adapting its image streams frees up network bandwidth that permits the other streams to be transmitted faster.

In comparison, the dashed line represents the non-adaptive case, again exhibiting undesirable PII slowdown.

In Figure 6.5, downsampling is applied to all of the image streams, not just one, and the measurements shown are the average frame rate experienced by the three streams. The non-adaptive scenario experiences a marked decrease in frame rate, while the frame rate for the adaptive version increases a good bit, due mostly to its reduced bandwidth usage.

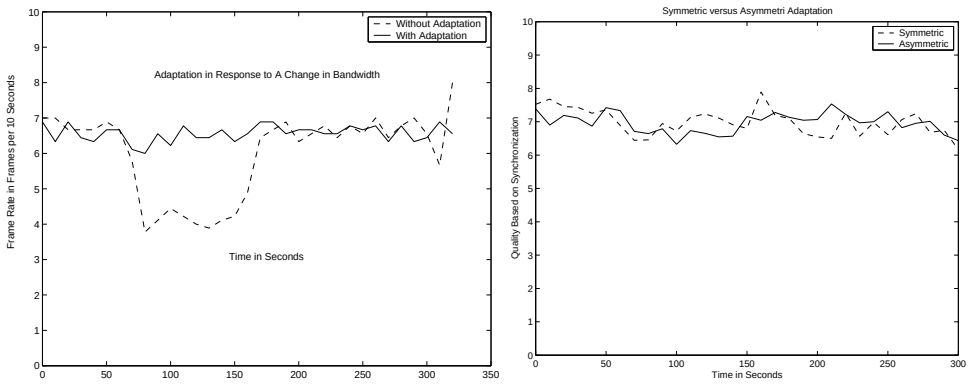

Fig. 3. Left: Adaptation Effects on Communication Bandwidth., Right: Quality Function Usage: Focus on 'Fresh' Information.

6.5 Using the Information Quality Metric

Synchronization Windows. The next experiments demonstrate the importance of formulating and using appropriate quality functions. In the first experiment shown in Figure 6.5, the quality function is formulated to prevent undue penalties incurred by any single client. This is achieved by placing additional weight on the synchronization window experienced by clients. The idea, of course, is to keep clients 'in sync', that is, to maintain for all clients reasonable levels of information 'freshness'[19]. This is particularly important in applications where recent information is more critical than high quality, older data. In the actual experiment, the quality function used prompts the optimization algorithm to favor a symmetric downsampling approach (i.e., all streams are downsampled) vs. an asymmetric approach (i.e., only one stream is downsampled).

Throughput. In contrast to the previous experiment, Figure 6.5 uses a quality function that focuses on the total throughput attained, measured as the number of kilobytes sent. In this case, asymmetric adaptations are favored over symmetric ones between times 90 and 150 seconds, which is when cross-traffic is present.

Alternative Adaptations. The final results presented in Figure 6.5 demonstrate the efficacy of alternative adaptation methods. The non-adaptive scenario (dashed line) is contrasted with an adaptive scenario (solid line) in which downsampling is done by reducing image color depth, from 24-bit color to 8-bit grayscale. This results in the total transfer of more potentially useful information to clients, but is less effective in preventing total slowdown.

Discussion of Results. In the scenario-based adaptations shown in this section, middleware-level source-based filtering *continuously* reacts to changes in available resources and in application needs, in order to guarantee desired levels

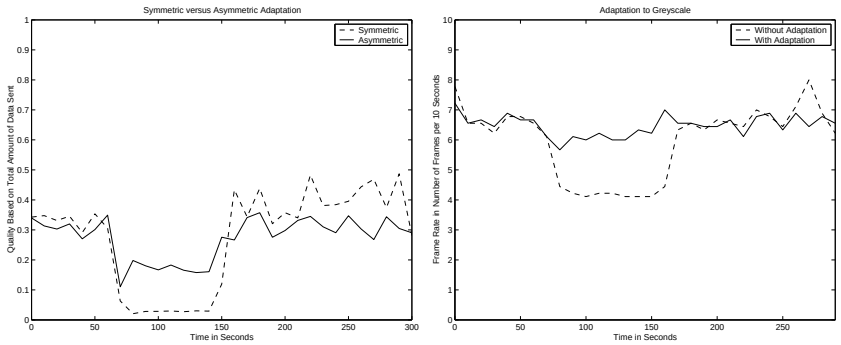

Fig. 4. Left: Quality Function Usage: Focus on Total Throughput., Right: Alternative Source-based Filtering: Color Downsampling.

of service and information quality. These reactions synchronize multiple data streams without the use explicit synchronization or control methods. Limitations to our approach include the following. First, application-specific methods for data filtering must be profiled in terms of the effects on information quality vs. resource usage and in general, such methods will experience limits in the tradeoffs they offer to quality management algorithms. Second, the additional costs incurred from continuous quality management may not be suitable for very resource-poor target systems. Third, better solution algorithms may be required to attain solutions scalable to thousands of participants and multiple concurrent applications. Finally, additional OS support is required in order to enforce participation in quality management by all applications and end users [28].

7 Conclusions and Future Work

This paper has shown dynamic adaptation based on information quality to be a viable solution for data stream interference in wireless systems. Two specific problems, sensor and client rate synchronization, are solved, and experimental results are attained on representative wireless platforms. While the adaptation algorithm requires $O(n \lg n)$ time, where 'n' is the number of clients, its actual runtime delay is shown acceptable even for larger problems (i.e., 100 clients).

Our future work will follow two directions. The first direction is to develop system- and middleware-level support for pervasive applications, with initial results described in [29, 3] and current efforts focused on the creation of kernel-level mechanisms for COTS Linux kernels. The second direction is to precisely define and solve other problems concerning information sharing in wireless environments, driven by realistic applications and constraints. One such problem is the dynamic selection of suitable sensors when clients are faced with choices in different sensors' delays vs. information qualities. A concrete instance of the

problem is how to track people in settings covered by multiple cameras. An interesting constraint concerns the power used by applications in embedded systems, particularly for handheld devices. The solutions we seek favor the continuous management of information quality over the provision of explicit mechanisms for stream management.

References

1. Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *Proc. of the Real-Time Systems Symposium*, Austin, Texas, November 2002.
2. Ralf Ackermann, Manuel Goertz, Martin Karsten, and Ralf Steinmetz. Prototyping a PDA based Communication Appliance. In *Proceedings of Softcom 2001, Split*, October 2001.
3. Sandip Agrawala, Chris Poellabauer, Jiantao Kong, Karsten Schwan, and Matthew Wolf. Resource-aware stream management with the customizable dproc distributed monitoring mechanisms. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003.
4. D.G. Andersen, H. Balakrishnan, M.F. Kaashoek, and R. Morris. Resilient Overlay Networks. *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 131–145, 2001.
5. Guruduth Banavan, Marc Kaplan, Robert E. Strom, Daniel C. Sturuman, and Wei Tao. Information flow based event distribution middleware. In *Middleware Workshop at the International Conference on Distributed Computing Systems*, 1999.
6. Thomas Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.
7. Scott Brandt, Gary Nutt, Toby Berk, and James Mankovich. A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage. *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 307–317, 1998.
8. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland, Oregon, July 2000.
9. Yuan Chen, Karsten Schwan, and David Rosen. Java mirrors: Building blocks for remote interaction. In *The 2002 International Parallel and Distributed Processing Symposium (IPDPS 2002)*, April 2002.
10. Yuan Chen, Karsten Schwan, and Dong Zhou. Opportunistic channels: Mobility-aware event delivery. In *ACM/IFIP/USENIX International Middleware Conference*, 2003.
11. Greg Eisenhower. The echo event delivery system. Technical Report GIT-CC-99-08, Georgia Institute of Technology, 1999.
12. Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *Symposium on Operating Systems Principles*, pages 48–63, 1999.
13. Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997.
14. Ashvin Goel, Molly H. Shor, Jonathon Walpole, David Steere, and Calton Pu. Using feedback control for a network and cpu resource management application. In *Proceedings of the 2001 American Control Conference*, June 2001.

15. Ching-Chih Han and K. G. Shin. Message transmission with timing constraints in ring networks. *IEEE Real-Time Systems Symposium*, 17:165–174, December 1996.
16. Qi He and Karsten Schwan. Iq-rudp: Coordinating application adapatation with network transport. In *High Performance Distributed Computing (HPDC-11), ACM/IEEE*, July 2002.
17. J. Huang, Y. Wang, and F. Cao. On developing distributed middleware services for qos- and criticality-based resource negotiation and adaptation. In *Journal of Real-Time Systems*, 1998.
18. T. Hudson, M. C. Weigle, K. Jeffay, and R. M. Taylor II. Experiments in best-effort multimedia networking for a distributed virtual environment. In *Proceedings Mutlimedia Computing and Networking*, January 2001.
19. Rainer Koster, Andrew Black, Jie Huang, Jonathon Walpole, and Calton Pu. Infopipes for composing distributed information flows. In *Proceedings of the ACM Multimedia Workshop on Multimedia Middleware*, October 2001.
20. Charles Krasic and Jonathan Walpole. QoS scalability for streamed media delivery. Technical Report CSE-99-011, Oregon Graduate Institute of Science and Technology, 17, 1999.
21. R. Kravets, K. Calvert, and K. Schwan. Payoff adaptation of communication for distributed interactive applications. In *The Journal for High Speed Networking: Special Issue on Multimedia Networking*, 1999.
22. B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. In *IEEE Journal of Selected Areas in Communication*, September 1999.
23. C. Lu, J. Stankovic, G. Tao, and S. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems Journal*, 23(1):85–126, July 2002.
24. Lui Sha and Xue Liu and Terek Abdelzaher. Queueing model based network server performance control. In *Real-Time Systems Symposium*, Austin, Texas, December 2002.
25. Dylan McNamee, Charles Krasic, Kang Li, Ashvin Goel, David Steere, and Jonathan Walpole. Control challenges in multi-level adaptive video streaming. In *Proceedings of the 39th IEEE Conference on Decision and Control (CDC 2000)*, December 2000.
26. Marija Mikic-Rikic and Nend Medvidovic. Architecture-level support for software component deployment in resource constrained environments. *First International IFIP/ACM Working Conference on Component Deployment*, June 2002. Berlin, Germany.
27. Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Symposium on Operating Systems Principles*, 1997.
28. Chris Poellabauer, Hasan Abbasi, and Karsten Schwan. Cooperative run-time management of adaptive applications and distributed resources. In *Proceedings of the 10th ACM Multimedia Conference*, December 2002.
29. Christian Poellabauer and Karsten Schwan. Kernel support for the event-based cooperation of distributed resource managers. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, September 2002.
30. Christian Poellabauer and Karsten Schwan. Power-aware video decoding using real-time event handlers. In *5th International Workshop on Wireless Mobile Multimedia (WoWMoM)*, September 2002.

31. Daniela Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On adaptive resource allocation for complex real-time applications. Technical Report GIT-CC-97-26, Georgia Institute of Technology, 1997.
32. Cyrus Shahabi, Roger Zimmermann, Kun Fu, and Shu-Yuen Didi Yao. Yima: A second generation continuous media server. *IEEE Computer*, pages 56–64, June 2002.
33. Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh-Based Content Routing using XML. *Proceedings of the Eighteeenth ACM Symposium on Operating Systems Principles*, pages 160–173, 2001.
34. Carlo Tomasi and Takeo Kanade. Shape and Motion from Image Streams: A Factorization Method Part 2. Detection and Trcking of Point Features. Technical Report CMU-CS-91-132, Carnegie Mellon University, April 1991.
35. Noanbor Wang, Douglas C. Schmidt, Anirudda Gokhale, Christopher D. Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall, and Richard E. Schantz. Total quality of service provisioning in middleware and applications. In *The Journal of Microprocessors and Microsystems*, volume 26, January 2003.
36. Rich West and Chris Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *IEEE Real-Time Systems Symposium (RTSS)*, 2000.
37. Matt Wolf, Zhongtang Cai, Weiyun Huang, and Karsten Schwan. Smart Pointers: Personalize Scientific Data Portals in your Hand. In *ACM Supercomputing Conference*, 2002.
38. D. Zhou, K. Schwan, G. Eisenhauer, and Y. Chen. Jecho - interactive high performance computing with java event channels. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2001)*, April 2001.

A Algorithm Pseudocode

Below is the pseudocode for a heuristic to solve the Synchronized Client Rate Problem. It runs in $O(n \lg(n))$ time, where n is the number of sensors. First, some helper functions are shown.

Quality q consists of wi, hi, fr, cb, cp, la. FUNCTION Try-Position takes a Quality, q, a sensor number i, a float representing bandwidth, and a float representing CPU. Try-Position returns 1 if the quality is 0.0, 2 if the quality is above zero, but too much sensor CPU is used, 3 if the sensor CPU load is acceptable and the above is good, but the bandwidth is bad. If the bandwidth consumption is acceptable, but the sensor CPU drain is not, then Try-Position returns 4. In any other case, Try-Position returns 5.

Quality-func is assumed to take constant time, i.e. it does not vary with the number of sensors. Sensor-Cpu-Func returns the amount of CPU timequality q takes for sensor i. Sensor-Bw-Func gives the amount of bandwidth quality q is expected to take up. Server-Cpu-Func describes how much sensor CPU a certain quality takes up. Try-Position gives an idea of the 'goodness' of a certain quality for one sensor, assuming that it should not use more than bw bandwidth and sensor CPU. All of the functions being called are assumed to take constant time.

FUNCTION Quality-Change(int up, int which, int sensor, Quality q, Quality-Range qrp, double bw-avg, double server-avg, int this-time, double *amount-off, int gr anularity, Quality-Ptr q-all)

If the value of up is 0, then overall quality is being increased. Otherwise, overall quality is being decreased. 'Which' ranges from one to 6 and determines the dimension that is being changed. Qrp contains min and max values for each quality dimension. Granularity determines by how much each quality dimension should change. For instance, if granularity is 1, one moves halfway from the current value to the extreme. For each increase of 1 in granularity, the amount of movement of the value toward the extreme in the dimension given by which halves.

Try-Position is called on the new position, and if the value returned by Try-Position increases or if the value returned by Try-Position is between 2 and 4, and the output of the function Get-Amount-Off is less than is was last time this function was called for this sensor, then the new value is kept. Otherwise, it is rejected. Get-Amount-Off returns how much more a certain quality would have to be reduced in order to achieve the next higher number in Try-Position. Note that Get-Amount-Off runs in constant time.

FUNCTION Synchronized-Client-Rate-Problem-Constraint-Solution returns Integer

For each client i, set $qrp.MIN$ and $qrp.MAX$ for each of the dimensionsto the absolute minimum value and absolute maximum value for that dimension, respectively.

Next, set the default values for the quality for each sensor and each dimension to the average of the minimum value for that dimension and the maximum value for any sensor for that dimension.

Below is the pseudocode for a heuristic to solve the Synchronized Client Rate Problem. It runs in $O(n \lg(n))$ time, where n is the number of sensors. First, some helper functions will be shown.

```
// server-avg is the amount of sensor
// time each client can use.
// It is one minus the current CPU drain
// divided by the number of sensors.

d =
The bandwidth drain
or this timestep
summed over all clients.
sensor-ok[i] for each sensor,i, is 0.
// Sensors start off having
// their constraints not
// satisfied.
// Server-slack and bw-slack
// are used later,
// and they start off at 0.

LINE A
for each sensor i
  last-time = 0
  // What was the result of Try-Position on
```

```
 // the current position on the last iteration?
 rep = 0
amount-off = 1.0
while (this sensor is not OK)
  if (this-time = last-time) rep = rep + 1
  select case (this-time)



  CASE 1:
    \\ Quality for this sensor is 0.0.
    \\ Should raise Quality first.
    copy-quality(q-temp, q[i])
    if (this-time = last-time)
      copy-quality(q-temp, q[i])
      q-temp.wi += 20, and
      q-temp.hi += 20
    // MAX.wi is the highest allowed
    // width, while MAX.hi is
    // the highest allowed height.
    if (q-temp.wi > MAX.wi)
          q-temp.wi = MAX.wi
      if (q-temp.hi > MAX.hi)
          q-temp.hi = MAX.hi
    q-temp.cp = No Compression
        // Try out the position resulting
//  from increases to wi and hi.
        j = try-position(q-temp, i, bw-avg,
        server-avg, q)
  // Update quality if this succeeds.
          if (j > this-time) then
            this-time=j and
            copy-quality(q[i], q-temp)
          else
        // Try out some increases and
        // decreases in a single dimension.
        // The actual change in quality is
// logarithmic in i.
            for i = 1 to 6
// Go through all of the reps
// until a good solution is
// found.
             if (quality-change(1, k, i,
               q[i], qrp[i], bw-avg,
               server-avg,
               this-time, amount-off
               rep, q)) break.
             if (quality-change(0, k,
                 i, q[i], qrp[i],
                bw-avg, server-avg,
```

```
                this-time, amount-off,
                rep, q)) break.
              copy-quality(q-temp, q[i])
     last-time = this-time



    CASE else:
    // Quality is good, but some
    // resource constraints are not met.
      copy-quality(q-temp, q[i])
      // Try downsampling without
      // worsening the situation.
      for k = 1 to 6
  quality-change(0, k, i,
          q[i], qrp[i], bw-avg,
          server-avg, this-time,
          amount-off, rep, q)
          then break.
      // If you can, then break.
      copy-quality(q-temp, q[i])
      last-time = this-time
// Only try a certain number of
// reps to make it all work.
// The limited number of repetitions
// keeps the heuristic fast.
    if ((this-time = 5) and (rep > 0)) then
        this sensor is OK
    else if ((this-time < 5) and (rep > 7))
        then this sensor not OK and
    last-time = this-time
LINE B

server-slack = 0.0
bw-slack = 0.0

// Find sensors that aren't using
// their share of the CPU and bandwidth,
// and then distribute to the others.

for each sensor i
  if this sensor is OK
    bw-slack += bw-avg
          - sensor-bw-func(q[i])
    server-slack += server-avg
          - server-cpu-func(q[i])
j = # sensors that are not OK

bw-avg += (bw-slack / j)
server-avg += (server-slack / j)
```

```
// Now, the algorithm is run again
// with the extra "slack" included.
Now, repeat everything between
LINE A and LINE B verbatim.
// Only the sensors that have
// not had their constraints met
// are run again.
// If even one sensor has unmet
// constraints, then failure has
// occurred.
for each sensor i
  if (this sensor is not OK) return 0
return 1
```

This is the code to solve the constraint-based version of the Synchronized Client Rate Problem. Most of the algorithm is linear with respect to the number of clients. The major exceptions are the two main loops. They run in $O(n \lg(m))$ time where m is the biggest difference between the maximum and the minimum of any of the six components of information quality and n is the number of sensors. The loop for each client will only go around a fixed number of times, because the heuristic will give up eventually. The number of times the while loop with the case statement goes around can not be more than 7 reps * 5 levels of how good one is * the number of clients. All of this must be multiplied by two because of "round 2".