A Distributed Building Evacuation System

by

Dany M. Qumsiyeh

S.B., Massachusetts Institute of Technology (2007)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

Author . . . .
Department of Electrical Engineering and Computer Science
June 25, 2008

Certified by . . . .
Jacob S. Beal
Postdoctoral Associate
Thesis Supervisor

Certified by . . . .
Gerald J. Sussman
Panasonic Professor of Electrical Engineering
Thesis Co-Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Distributed Building Evacuation System

by

Dany M. Qumsiyeh

Submitted to the Department of Electrical Engineering and Computer Science
on June 25, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis investigates the feasibility of a smart building evacuation system, capable of guiding occupants along safe paths to exits and responding to changing threats. Inspired by developments in amorphous computing, the design presented is scalable to large networks, robust to hardware and communication failure, and based on simple low-cost components. A simulation and hardware prototype demonstrate that this distributed building evacuation system is both feasible and cost effective.

Thesis Supervisor: Jacob S. Beal
Title: Postdoctoral Associate

Thesis Co-Supervisor: Gerald J. Sussman
Title: Panasonic Professor of Electrical Engineering

Acknowledgments

I would like to thank:

My advisor, Jake Beal, for forcing me to settle on a topic, work on the topic that I chose, and write about what actually I did, even when I vehemently opposed the idea. His feedback was indispensable.

Gerry Sussman, for making everything seem okay. Jonathan Bachrach and Bill Butera, for their inspiring work and guidance at the beginning of the project. Piotr Mitros, for reminding me how control systems work. Mark Tobenkin, for always tempting me with interesting problems.

My parents, for supporting me in everything that I do. My close friends, who have cared about me and kept me going. And all the rest at TEP and East Campus, for distracting me from my thesis in the best ways possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Imagine an emergency system capable of guiding building occupants not just to the nearest exit, but along the safest route, avoiding fire and damaged areas. Such systems have been proposed before, but have daunting reliability issues: how likely is it for a system failure to leave occupants with no directions out of a building, or—worse yet—lead them in the wrong direction? Many patents describe smart evacuation systems involving a central controller, where damage to a single point in the building could undermine the entire system. Research in sensor networks promises much better *distributed* solutions, where many smart nodes are connected across a building, so that even when large areas are damaged, any remaining nodes continue to function.

I investigated an extreme in this design space: a network of very cheap and small nodes, densely covering all areas of a building. This thesis focuses on algorithms for smart guidance in such a system. I demonstrate the feasibility of a dense, distributed evacuation system by showing that algorithms for dynamic guidance can be simple, require very little hardware, and scale to very large networks.

If it is feasible to embed nodes in every ceiling or floor tile, then one might also imagine computing elements small enough and cheap enough to be embedded into paints and everyday materials, as suggested in paintable computing[6]. These programmable elements could create smart materials that sense or affect their environment, like a paintable display or a bridge that monitors its own condition. By exploring the idea of an evacuation system built into every floor tile, this research also hopes to contribute to the compelling vision of paintable computing.

1.1 Previous Work

This section briefly describes standard fire evacuation systems and previously proposed improvements. The next section then describes the system that I propose and analyze in the remainder of the thesis.

Standard fire alarm systems[14] consist of smoke detectors and pull stations connected to a central panel that triggers the alarm and provides status information. Strobe lights and sirens alert building occupants, and lighted exit signs over doorways and in hallways guide occupants to the nearest exit. The alarm system, emergency lights, and exit signs may each have backup batteries to allow operation during a power failure. Alert devices and smoke detectors are often connected in chains of multiple devices, or loops that can tolerate a single broken connection, but still present the possibility that local damage to wiring—and especially the control panel—can render many devices ineffective.

Many patents describe more intelligent egress systems, replacing exit signs with indicators that can be controlled to redirect occupants. An early patent[15] suggests sequential activation of lights and buzzers to direct people in the presence of smoke or noise. Many other patents suggest central control systems that can receive sensor input and redirect occupants using sequences of lights[4], light strips[21], tape with embedded LEDs[20], exit and no-exit signs[12], or arrows[5].

These and many other patents (e.g. [16], [17]) all suggest systems of devices connected to a central computer or control circuit, much like modern alarm panels. This central device and the connections to it are a weak point where local damage might render all occupants without guidance to an exit.

In the field of sensor networks, many systems have been developed that can operate effectively despite the loss of a few devices. This robustness is especially useful in situations with harsh conditions, such as military or emergency applications. Building evacuation is frequently mentioned in sensor network literature as a possible application. For example, [11] outlines emergency situations that can benefit from sensor networks and cites the Smart Signs system[9], which uses a wireless network of display devices and personal tags to guide users to various destinations, including exits in case of an emergency.

The Berkeley Fire Information and Rescue Equipment (FIRE) project[19] prototyped a distributed system for providing information to firefighters and enhancing their communication. It includes a subsystem called SmokeNet, a wireless sensor network that could be used for building evacuation. One description describes a "stoplight" beacon that would be placed on both sides of every doorway to indicate safe pathways. Such systems could be designed to operate effectively despite the loss of a few nodes, providing a significant advantage over centralized systems.

Both the Smart Signs system and SmokeNet describe networks where devices are installed at particular locations, like doorways, and have a large display or beacon for indicating routes. These devices carry out complex tasks like message routing, and in the Smart Signs system, computing individual paths for each user. This thesis explores a different design choice: having much simpler devices in greater numbers covering every space in a building.

1.2 Proposal

I explored the feasibility of a distributed and *dense* network of devices for providing active guidance in a building fire. If the devices were cheap enough and small enough to be embedded in every ceiling tile or floor tile, or woven into carpet, such a system would provide unprecedented building coverage, redundancy, signal visibility, and sensor information. The proposed system has thousands of these small computational devices distributed and networked across a building. Each one has feedback, sensing and communication capabilities, and a control device such as a microcontroller, configurable logic, or a custom digital circuit.

To facilitate low cost and high volume production, nodes are identical, have few components, and allow for wide performance variability. The system must be very forgiving of node failures and manufacturing variations.

The nodes communicate locally to a small number of neighbors, simplifying communication overhead and interconnect complexity. As often the case in sensor networks, nodes could communicate wirelessly, but in my analysis I assume wired connections. The floor, ceiling, or carpet tiles of such a system might be designed to allow conductivity between neighboring tiles, so that connections need not be wired manually.

Nodes are connected only along building pathways and through open areas, so that the building topology need not be programmed in. The system assumes that a communication path from one node to another is also a feasible path for occupants. Exits are indicated explicitly—via specially chosen nodes or a tag such as an electrical jumper.

Each node has an indicator, such as one or more LEDs, and a small sensor, such as a thermistor. The sensor will be used to both trigger the alarm, and measure the threat when determining exit pathways. As these devices might be made much smaller than a single exit sign or directional indicator, I explored the idea that the devices instead cooperate to produce large-scale visual signals. Large, sweeping patterns of light may be a promising alternative to exit signs, providing high visibility and little ambiguity. Much of this thesis focuses on algorithms for producing coherent sequences of light among many devices.

The design does not preclude input from pull-boxes for manual alarm triggering, or smoke detectors and other traditional fire alarm equipment. Many other considerations such as packaging, mounting, physical robustness and power distribution are also important to the design of evacuation systems, but will not be covered. This thesis simply demonstrates the feasibility of such a system by showing that algorithms for dynamic guidance can be simple, require very little hardware, and scale to very large networks.

1.3 Outline

Chapter 2 shows how the *self-healing gradient*, a common building block in sensor networks and amorphous computing, can determine safe paths towards exits and adapt to changing network conditions. Chapter 3 then investigates how to produce moving patterns of light along these paths, by having each node track the state of the next node in the path. I show how damping is important for reducing errors in the system, and how a strategy similar to phase-locked loops reduces the effects of intermittent communication. Chapter 4 finally describes the prototype and system simulation used to verify the algorithms and show that they are effective at building evacuation.

Chapter 2

Self-Healing Gradient

The self-healing gradient is a common building block in sensor networks and amorphous computing[1] that approximates distances in a network. This chapter first describes the algorithm and some useful variants, then explains how the spanning tree produced by the gradient can be used to direct occupants towards exits.

2.1 Review of Gradient Algorithms

Gradient is an iterative algorithm that is evaluated on every node and computes the distance[1] in the network from each node to the nearest designated *source* node. The "distance" computed might be a hop count, shortest path distance, or other metric, but is often used as an approximation for geometric distance (for example, to establish coordinate systems in a sensor network[2]). If nodes in an evacuation system are installed at evenly spaced intervals, a simple hop count may suffice to determine which exit is closer.

Figure 2-1 illustrates the evaluation of a gradient algorithm on nodes arranged in a narrow space, like a hallway. The leftmost and rightmost nodes have been marked

[1]Note that the value computed is a scalar, rather than the vector field implied by a mathematical gradient. The name actually refers to chemical gradients, which are smooth changes in concentration over distance.

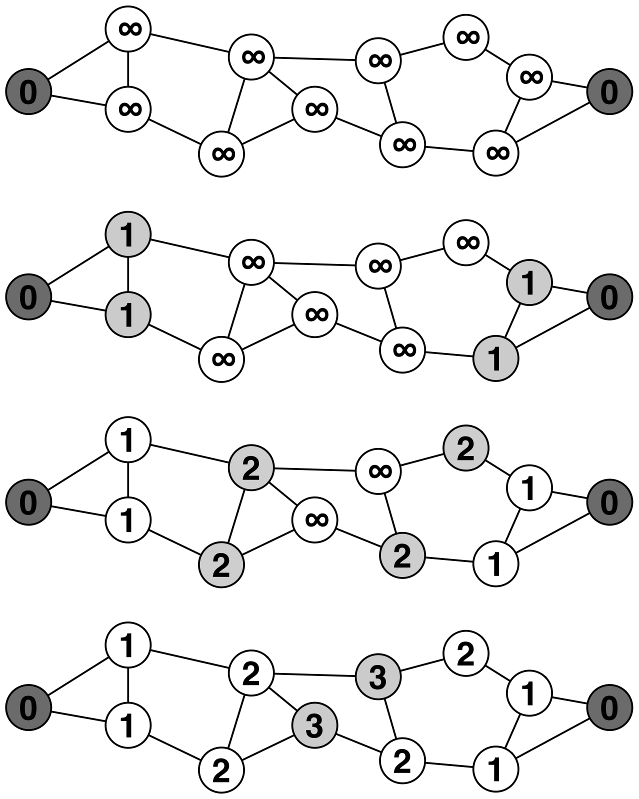

Figure 2-1: A gradient algorithm evaluated synchronously on a small network. Dark nodes are marked as sources. Changed nodes at each stage are light gray. Source nodes have value zero. At each iteration, every other node takes the minimum value among its neighbors and increments it, computing a hop-count to the nearest source.

as source nodes, representing exits. Every node carries a gradient value, and the algorithm iteratively updates this value based on the values of neighboring nodes. These updates may happen asynchronously in a real system, but in this illustration, all nodes are updated together. The source nodes always have a value of zero, and every other node is initialized with a large value, shown as infinity in the figure, indicating that no path to a source is known yet. At each iteration, a node updates its value to the minimum value among its neighbors, incremented by one. When the network is static, the gradient values converge to the distance in hop-counts of the nearest source.

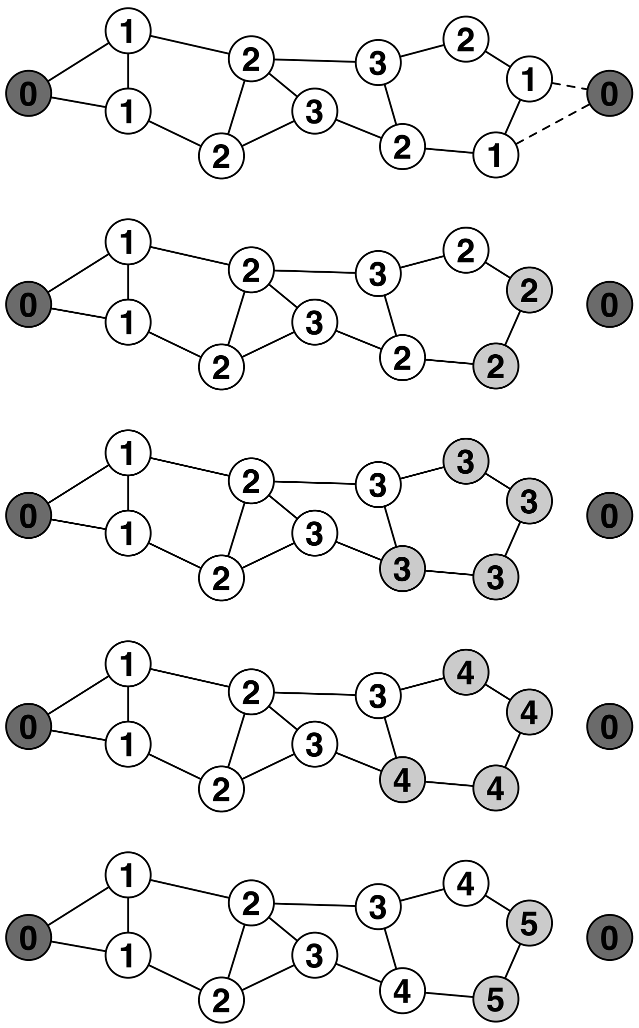

Figure 2-2: A gradient algorithm responding to a change in the network. The connections to the rightmost node are removed at the first time step. At each step, every node updates its value synchronously based on the previous values of its neighbors. The gradient repairs in $O(diameter)$ time.

The gradient algorithm described above is *self-healing*, in that it also adapts to changes in the network. Figure 2-2 shows what happens if the rightmost node is disconnected. In an amount of time proportional to the diameter of the network, the nodes update to represent the new distances to the source.

Many variations on this basic algorithm are applicable to building evacuation. If nodes are irregularly spaced, distance estimates at each hop can be incorporated to better approximate euclidean distance. In a wireless system, for example, signal strength could be used to estimate distances. Rather than *incrementing* the minimum value among its neighbors, each node would add the approximate distance to that neighbor to get a new gradient value.[2]

Threat avoidance algorithms[7] are another variation of gradient that incorporates a threat model to compute the *safest* path to a source. Rather than propagating a value representing distance, nodes can propagate a probability of survival that is multiplied along the path. Given a probability of survival at every node (based on sensor readings and a threat model), this algorithm can compute a maximum probability of survival path. Eames[7] even suggests that such an algorithm could find escape routes in a burning building.

2.2 The Gradient Spanning Tree

In the proposed evacuation system, each exit is marked with a single source node, and a threat avoidance-based gradient is used with temperature sensors to determine a probability of survival.

The phase tracking algorithms described in Chapter 3 do not use the gradient value directly, but rather the information about which neighbor had the minimum distance value (or maximum likelihood of survival). This indicates which neighbor is next along the safest path to the exit, and overall describes a spanning tree representing

[2]This form of gradient may converge very slowly in a self-healing context, due to a problem with closely spaced nodes. Beal et al. described this *rising-value problem* and presented the CRF-Gradient, a variation that provably reconfigures in $O(diameter)$ time[3].

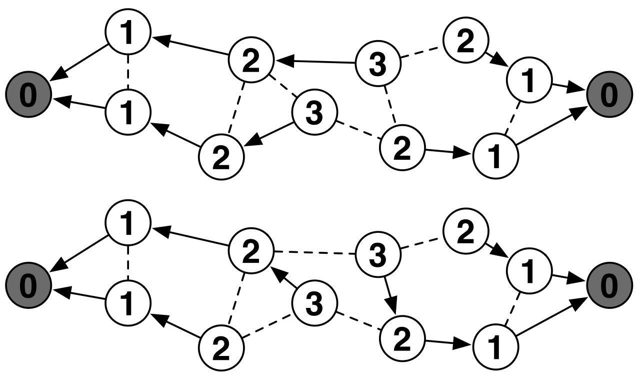

Figure 2-3: Spanning trees produced by gradient propagation. From each node, an arrow is drawn towards the neighbor with minimum gradient value. Ties are broken arbitrarily, so in this network there are many possible spanning trees, two of which are shown.

the propagation of the gradient, as shown in Figure 2-3. If there is a tie between two neighbors, either may be chosen; the prototype implementation chooses the neighbor most recently updated.

Each node tracks the neighbor ahead of it to determine when to blink its light, such that the patterns of light flow towards the exits. Chapter 3 investigates the properties of the tracking algorithm that are necessary to ensure coherent patterns of light across very large networks.

Chapter 3

Phase Tracking

Given spanning tree information from the gradient, the evacuation system uses a strategy of *phase tracking* to produce coherent patterns of light. This chapter analyzes a number of possible phase tracking techniques, and shows how damping and frequency correction are important properties for producing patterns across very large networks.

Figure 3-1 shows a single row of nodes producing moving pulses of light. Each node blinks its light at a regular rate, and the blinking is offset by a constant amount at each node, producing moving pulses of light. The blinking is tracked at each node by a phase value, θ, which cycles from 0 to 2π. When θ is below some threshold, the light is turned on.

Each node tracks the phase of the node ahead of it—the neighbor closest to the exit or next along the safest path—and attempts to keep a fixed phase offset from its neighbor. Some seemingly straightforward algorithms for phase tracking do not perform well with unreliable communication and variations in clock rate, and create less coherent patterns of light in larger networks. I found that damping is important for reducing the effect of these phase errors; particularly, the tracking algorithm should provide damping on all frequencies. The strategy used in phase-locked loops satisfies this condition, and works well in producing stable patterns of light.

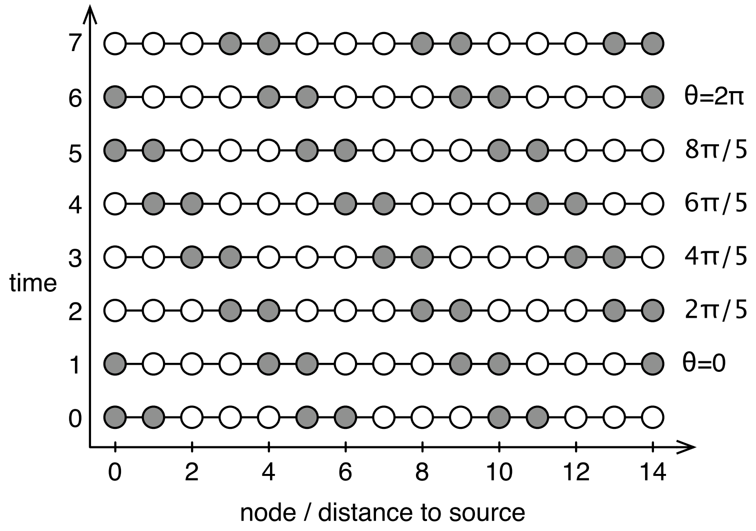

Figure 3-1: A row of nodes shown at many instants in time, producing moving pulses of light. The leftmost node is marked as a source, so the pulses move to the left. Each node pulses at a regular rate with a phase offset from its neighbor. Phase values are shown for the rightmost node.

The phase tracking problem is related to the problem of time synchronization. Section 3.1 first motivates the phase tracking strategy by comparing it to time synchronization techniques that might also solve the problem. Section 3.2 then explains the simulation methods and assumptions. Section 3.3 analyzes basic techniques for correcting phase directly, and demonstrates the need for damping. Section 3.4 analyzes feedback-based techniques for adjusting the frequency of the phase cycle instead, and shows why damping is necessary across all frequencies. These phase tracking techniques are analyzed using large-scale simulations, and in some cases verified on a small prototype system, described in Chapter 4.

3.1 Relationship to Time Synchronization

The strategy of phase tracking can be considered a form of time synchronization performed along the gradient spanning tree. Indeed, any solution to the problem of

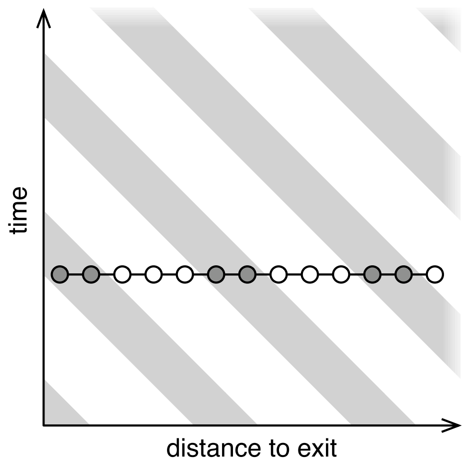

Figure 3-2: Moving patterns of light can also be described by a striped function of time and distance to the exit. A row of nodes is shown at one instant in time, and nodes that fall on a shaded stripe have their light on. An approach related to phase tracking could use time synchronization to determine the current time, a gradient to approximate the distance to the exit, and this striped function to determine light output.

time synchronization can be used to create the same patterns of light. If the current time is known, and the gradient approximates the distance to the exit, then the moving pulses of light can be defined by the striped function shown in Figure 3-2. This section compares the phase tracking solution to previous techniques for time synchronization.

Traditional approaches to time synchronization, such as the Network Time Protocol (NTP) and Global Positioning System (GPS), favor accuracy over simplicity and low cost. Sivrikaya and Yener[18] point out that the complexity and cost of these approaches makes them unsuitable for sensor networks. GPS requires expensive hardware, and would not work well indoors. NTP requires complex agreement algorithms and the infrastructure to route messages to a small number of time servers.

Mirollo and Strogatz[13] present a compelling decentralized algorithm for synchronization, inspired by the way fireflies synchronize their flashes. They prove that a simple behavior, where an oscillator is more likely to fire after observing pulses from other oscillators, will always converge to synchronized firing in a fully connected network. Lucarelli and Wang[10] showed that it also converges in multi-hop networks. Given the synchronized phase information from this algorithm, a phase offset could easily be computed from the gradient value to produce the desired pattern of lights. This strategy is promising, but its convergence time is potentially unbounded, and it may not converge quickly in a building evacuation scenario. Lucarelli and Wang show an *upper* bound on the rate of convergence that is proportional to the algebraic connectivity of the graph—a parameter that is large for dense graphs, but small for sparse graphs. A network of nodes arranged along hallways in a building, where many nodes are connected in long chains, would have a very small algebraic connectivity and may take a long time to converge. This algorithm may provide a good way to *keep* nodes synchronized, though, and should be explored further.

Greunen and Rabaey[22] propose lightweight tree-based synchronization (LTS) methods, where a spanning tree is constructed from a node with an accurate measure of time, and pairwise synchronizations are performed along the branches. This approach compromises between the expensive and centralized systems and the potentially slow distributed solution, and specifically targets large multi-hop networks like this evacuation system. However, the pairwise synchronization method that they propose is very similar to the undamped phase correction analyzed in Section 3.3.1, and may have consistency problems in large networks.

The proposed evacuation system uses a strategy similar in form to LTS, performing synchronization along a spanning tree, but has some differences. The gradients from the exits in this system imply separate spanning trees for each exit, but this is okay; coherent signals towards each exit are more important than global synchronization across the building. Also, this system performs synchronization continuously, whereas the LTS methods suggest separate stages for forming a spanning tree and performing synchronization. Synchronization is performed with every transmission, because the gradient may change constantly with sensor readings or broken connections.

The phase tracking algorithms studied could easily be converted to time synchronization algorithms by performing operations with an unwrapped phase[1], but the phase-based approach offers some advantages over using time synchronization:

- The cyclic phase value can be restricted to a small number, more easily handled and communicated by a small microcontroller. The prototype in Chapter 4, for example, uses a 16-bit phase value on an 8-bit microcontroller.

- The gradient value need not represent the distance to the exit. For example, threat avoidance algorithms use a value representing probability of survival, where the neighbor with the *highest* value represents the next node in the chain. Because a phase offset is introduced at every node, the phase tracking algorithm doesn't require a hop-count calculation.

3.2 Analysis and Simulation Methods

The algorithms below are tested by simulating a long chain of nodes synchronously, while modeling different sources of error. A simple chain of nodes provides a good characterization of the behavior of these algorithms because of the directional flow of information in a network: phase information always propagates outward from the source nodes. In a static gradient, then, each branch of the gradient spanning tree can be analyzed separately, and behaves the same way as an isolated chain of nodes.

The simulation models problems caused by clock error, but evaluates nodes in a globally synchronous manner. In a real system, nodes repeatedly process received information from neighbors and update their own values at a regular rate, but do so asynchronously between nodes. Because of clock rate differences, some nodes

[1] *Unwrapped phase* is a term used in signal processing, where the phase of a complex signal is plotted with values beyond 2π instead of cycling back to zero. For the algorithms in this chapter, phase is not part of a complex signal, and calculations with an unwrapped phase may behave differently than with a cyclic phase. The simulations use an unwrapped phase, but I assume that their behavior in the relevant regime is still indicative of the behavior of the cyclic version.

may update faster than others. In the simulation, all nodes are updated at once, and these asynchronous effects are approximated by scaling the values at each node appropriately. For example, phase values at each node would be incremented by different amounts based on the node's clock error.

There are many possible sources of error in an actual system. Except where noted, the simulations only model clock error and intermittent communication, suspected to be the primary challenges to algorithm scalability. In a physical system, problems may also be caused by integer rounding errors in calculation, effects from asynchronous updates, variation in clock rates over time, or variation in communication delays. However, the noise introduced by clock error and intermittent communication seems to be enough to characterize the phase tracking algorithms studied and their behavior in the face of uncertainty.

Clock error is modeled by choosing random clock rates for each node, uniformly distributed over $\pm 10\%$. This 10% bound reflects the specification for the internal RC oscillators used in the prototype (Chapter 4). Intermittent communication is modeled by allowing each node to receive data in a time step with only 1/3 reliability. This figure reflects the idea that each node multiplexes communication between three neighbors, as described in Section 4.1.2. However, these error rates are not intended to closely match the behavior of the prototype. Rather, they are used to determine whether the phase tracking algorithms degrade in performance over large networks, given *any* amount of error. This study is concerned with the scalability of the algorithms, independent of specific parameters or constants.

3.3 Phase Correction

One obvious phase tracking strategy is for each node to correct its own phase directly. Because clock rates differ between nodes, they will have to repeatedly perform this correction to avoid drifting out of sync. Below, I analyze a simple correction method, then show the advantage of damping on error propagation.

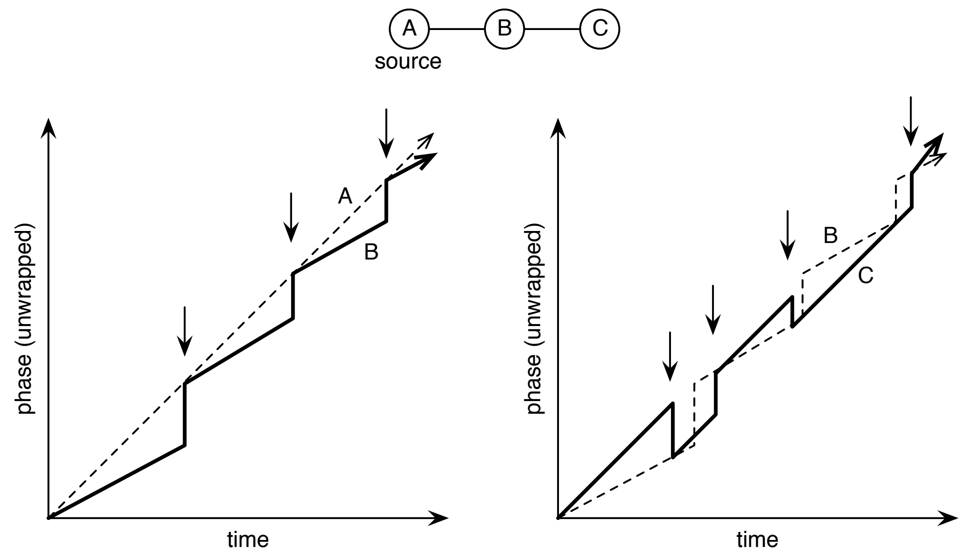

Figure 3-3: Three nodes in a chain using undamped phase correction to synchronize, where node B has a much slower clock than the others. Arrows mark times when messages are received. Irregular communication combined with clock variation causes phase jitter, which may grow worse farther away from the source.

3.3.1 Undamped Phase Correction

First, consider the simplest method of correcting a node's phase: each time a new value is received from the target neighbor, simply reset the phase to the new value, adjusted by the desired offset. Figure 3-3 shows an example plot of this undamped phase correction running on a few nodes with an exaggerated difference in clock rates. The plot shows how the combination of irregular communication and clock variations can cause phase jitter, which might disrupt the pattern of lights.

A real system wouldn't have such extreme variation in clock rates, but might have nodes that are hundreds of hops away from an exit. What happens when this effect is compounded many times over? Figure 3-4 shows the light output from a simulation of 300 nodes in a chain, revealing discontinuities in the pattern of lights.

Why do these discontinuities appear? Figure 3-5 shows the phase error from the same simulation, revealing the pattern of error propagation. One can almost follow

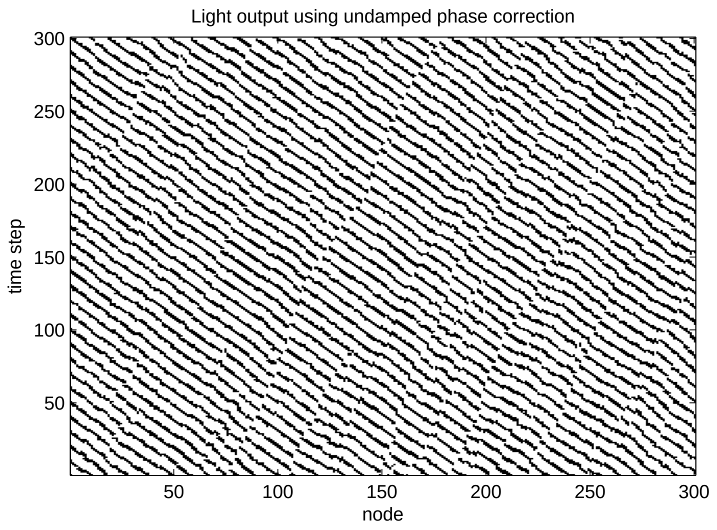

Figure 3-4: Light output from a simulation of undamped phase correction on a chain of 300 nodes, with the leftmost node (zero) marked as a source. Note that stripes move to the left, but frequent discontinuities appear farther from the source due to accumulated clock error.

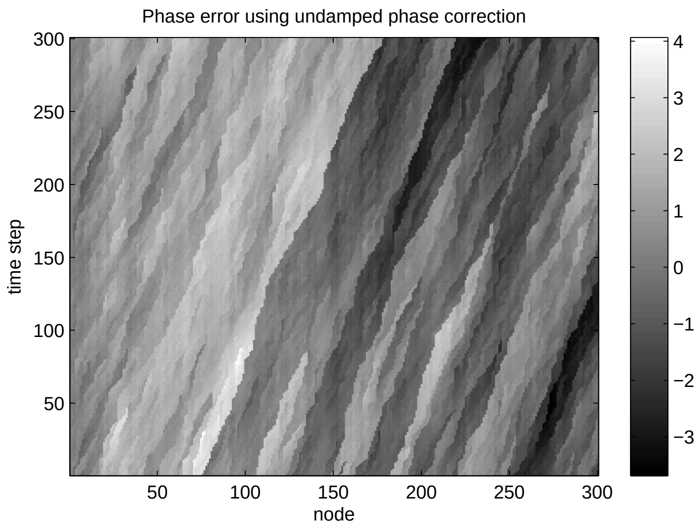

Figure 3-5: Unwrapped phase error for the simulation run in Figure 3-4. Note that errors tend to propagate outward without diminishing, causing steep discontinuities.

the path of a single phase value as it propagates; when values linger on nodes with slow clock rates, they create deepening valleys that continue down the chain, while other values drift in the opposite direction.

The variance of phase error provides a useful measure of the quality of the light pattern.[2] Figure 3-6 shows that this variance grows linearly with the distance to the source, indicating larger discontinuities.

The mean phase error indicates consistent offsets between one node and another. Figure 3-7 shows that the mean varies with trends in the clock deviations; many slow clocks along a path will cause the phase to lag behind.

The phase error plot in Figure 3-5 also reveals the rate at which phase updates propagate, indicating how quickly the system would respond to a change in the network. As expected from the 1/3 probability of communication each time step, changes propagate at about one node every three time steps.

3.3.2 Damped Phase Correction

Consider instead a system with some damping, where phase values gradually mix from one node to another. Conceptually, a node would take only a proportion K of its phase from the neighbor's phase, and the rest from its previous phase. Here, $\theta[n]$ is the node's phase at time n, $\phi[n]$ is the phase received from the neighbor, θ_{off} is the target phase offset from the neighbor, and $\Delta\theta$ is the default rate of progression (the rate if this node's clock were accurate):

$$\theta[n+1] = K(\phi[n] + \theta_{\text{off}}) + (1-K)\theta[n] + \Delta\theta \tag{3.1}$$

Thus, $K = 1$ represents an undamped system, and smaller values of K indicate greater damping. This equation doesn't work properly when the phase is cyclic, though. To ensure that correction is applied in the right direction for a cyclic phase,

[2] A low variance suggests a stable pattern, but note that a high variance does not always imply a noisy pattern; a slowly drifting phase may have high variance but still be acceptable.

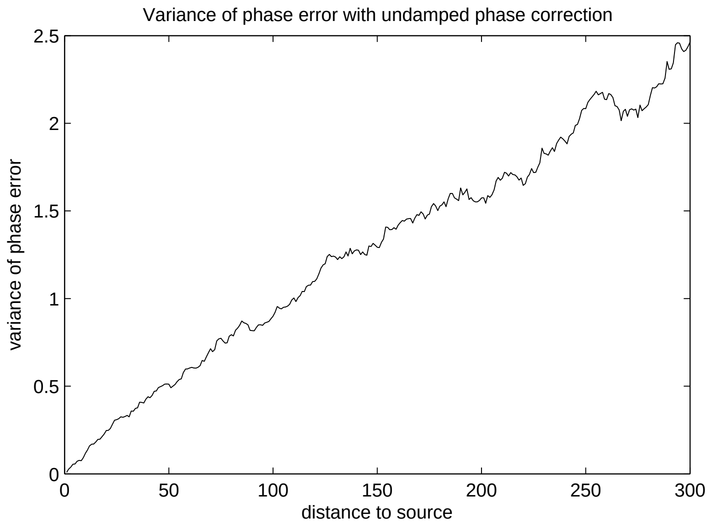

Figure 3-6: Variance of phase error using undamped phase correction, over 5,000 time steps. The variance grows linearly with the distance to the source, indicating that light patterns will become less coherent farther from the exit.

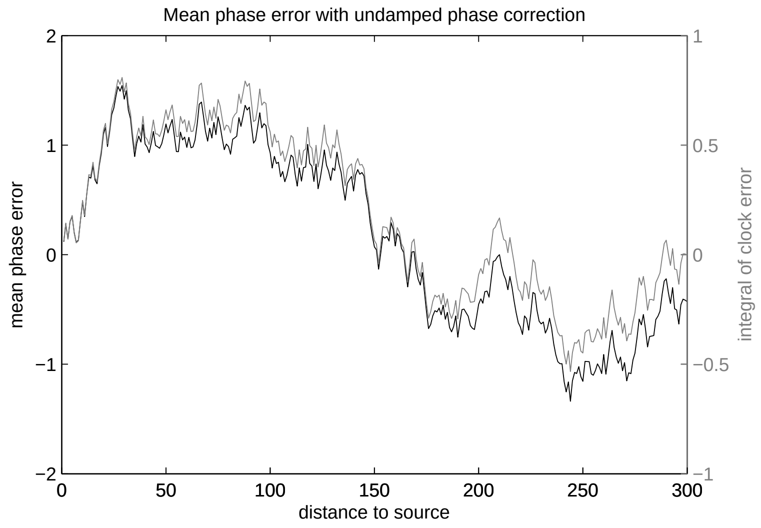

Figure 3-7: Mean phase error using undamped phase correction, averaged over 5,000 time steps. The mean closely follows the integral of the clock errors, which plots for each node n the sum of the clock deviations for nodes 1 through n.

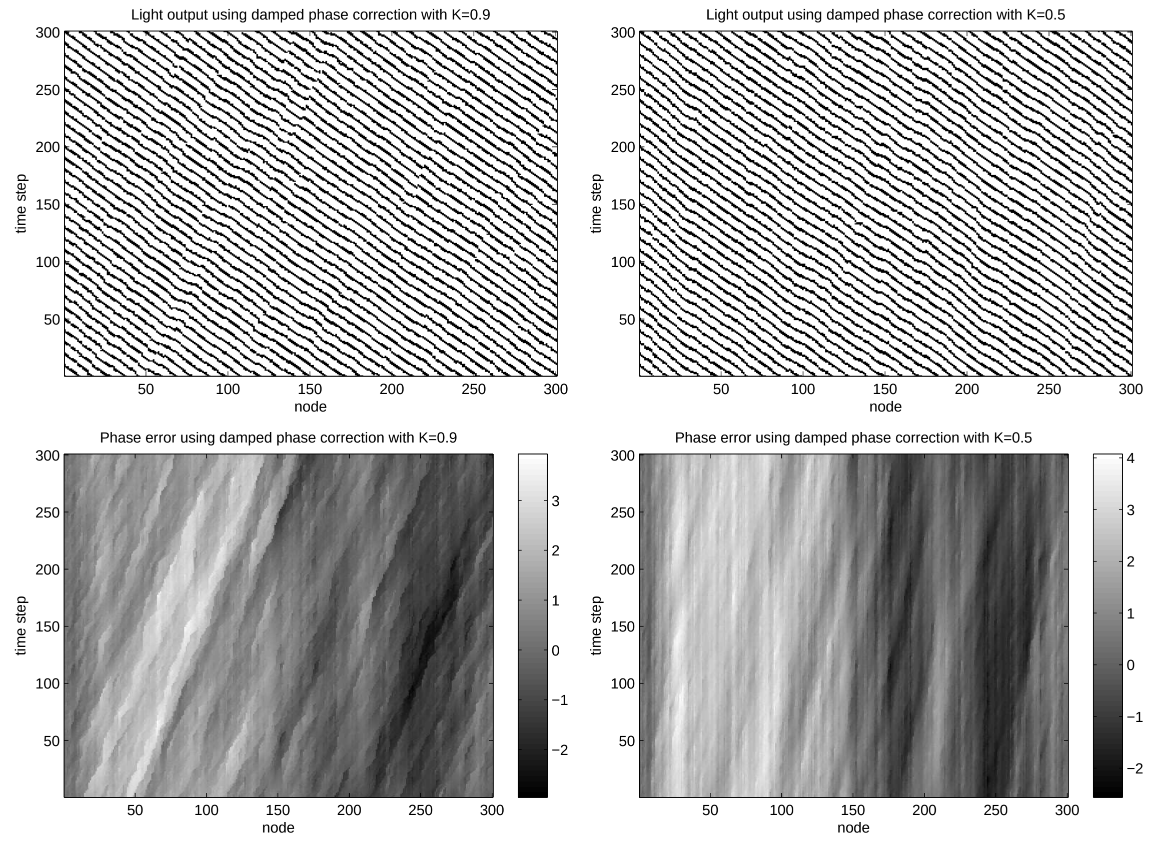

Figure 3-8: Light output and phase error using damped phase correction with $K = 0.9$ (left) and $K = 0.5$ (right). The damping causes phase values to blend together, reducing discontinuities.

we can factor out the phase error $\theta_{\text{err}}[n] = \phi[n] + \theta_{\text{off}} - \theta[n]$ and shift it by a multiple of 2π such that $-\pi \leq \theta_{\text{err}}[n] < \pi$. The update equation becomes:

$$\theta[n+1] = \theta[n] + \Delta\theta + K\theta_{\text{err}}[n] \tag{3.2}$$

Running two simulations of damped phase correction with $K = 0.9$ and $K = 0.5$ shows that even a little damping greatly reduces the appearance of discontinuities (Figure 3-8).

Figure 3-9 shows that the variance of phase error drops significantly, even for $K = 0.9$. The variance also seems to grow at a sub-linear rate. However, damping

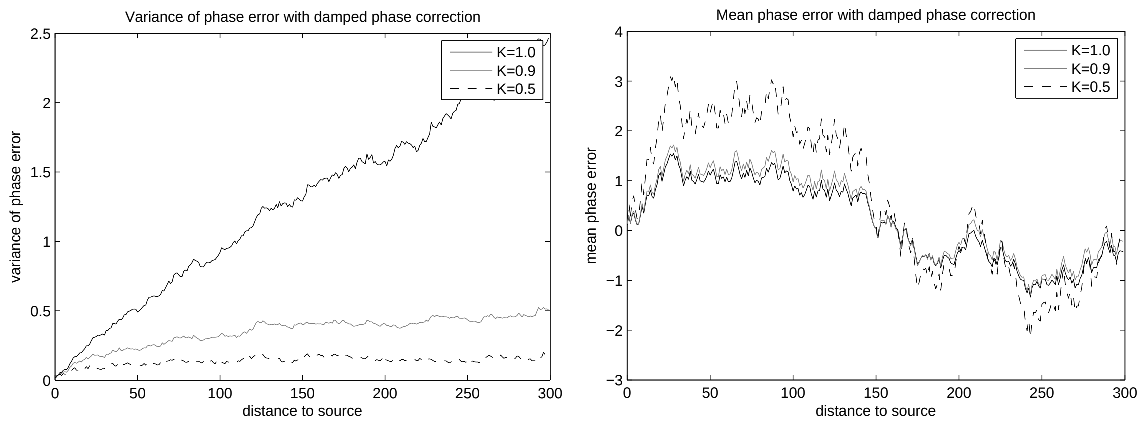

Figure 3-9: Mean and variance of phase error using damped phase correction, over 5,000 time steps. The variance decreases dramatically with damping, but the mean drifts farther at each node.

slows the propagation of phase values, allowing the mean phase error to drift farther. The large differences in mean phase error indicate that nodes are not achieving the desired phase offset from their neighbor.

The phase error plots in Figure 3-8 show that the propagation rate approximates Kp, where p is the communication probability: for $K = 0.5$, changes travel about one node every six time steps.

Over-damping

The variation in mean phase error suggests a lower bound on K: for a very small K, a slow node might lag so far behind it's neighbor that it blinks *after* the node closer to the source. This means the pulse might appear to be going in the wrong direction! Figure 3-10 shows an example of an over-damped system.

To find this lower bound, consider a slow node tracking a target phase $\phi[n]$, progressing at constant rate $\Delta\phi$. With a communication probability p, assume the average case where updates occur regularly with interval $1/p$. The phase lag developed

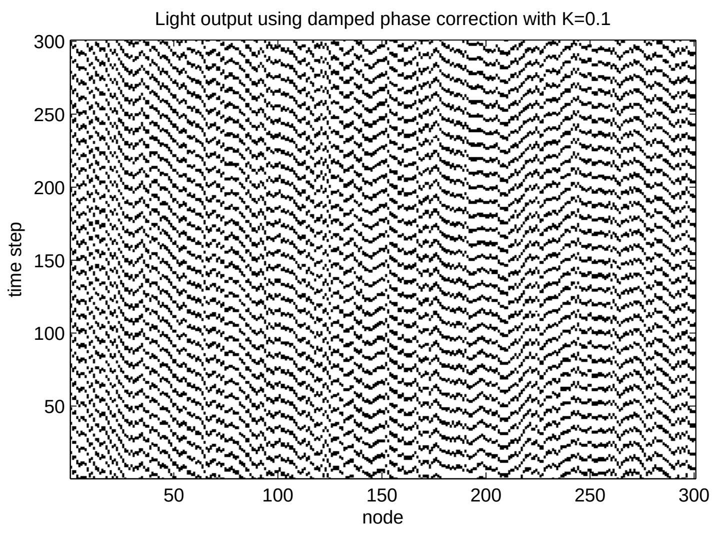

Figure 3-10: Light output using over-damped phase correction, where $K = 0.1$. Because phase values propagate so slowly, clock variations have enough influence to make the patterns go in the wrong direction.

during one update interval is:

$$\frac{\Delta\phi}{p} - \frac{\Delta\theta}{p} = \Delta\theta\frac{r-1}{p} \qquad r = \frac{\Delta\phi}{\Delta\theta} = \text{clock ratio } \frac{f_{\text{src}}}{f} \tag{3.3}$$

Assuming that the phase of every node is progressing steadily, the rate at every node should match the rate at the source, so the ratio r can be computed by dividing the source's clock frequency by that of the node in question. In steady state, the phase lag per update interval matches the correction applied by the update $(\theta[n+1] - \theta[n] - \Delta\theta)$:

$$\Delta\theta\frac{r-1}{p} = \theta[n+1] - \theta[n] - \Delta\theta \tag{3.4}$$

Using the update equation (3.2):

$$\Delta\theta\frac{r-1}{p} = K\theta_{\text{err}}[n] \tag{3.5}$$

We would like to ensure that $\theta_{\text{err}}[n] < \theta_{\text{off}}$ so that this node blinks before its neighbor:

$$
\begin{aligned}
\Delta\theta\frac{r-1}{Kp} &= \theta_{\text{err}}[n] \\
\Delta\theta\frac{r-1}{Kp} &< \theta_{\text{off}} \\
\frac{\Delta\theta}{\theta_{\text{off}}}\frac{r-1}{p} &< K \qquad (3.6)
\end{aligned}
$$

This provides a lower bound on K. In the simulation, for example, we have $p = 1/3$, $\Delta\theta = \pi/5$, $\theta_{\text{off}} = \pi/5$, and in the worst case $r = 1.1/0.9$. Thus, we should prefer $K > 2/3$. Also, note that the quantity $\Delta\theta/\theta_{\text{off}}$ represents the target speed of the pulses in nodes per time step: the number of nodes between pulses is $2\pi/\theta_{\text{off}}$, and the target number of time steps per cycle is $2\pi/\Delta\theta$. Thus, the equation also represents an upper bound on the pulse speed: even with $K = 1$, nodes producing fast-moving pulses might lag far enough in an update interval to blink at the wrong time.

3.4 Frequency Correction

The phase jitter seen in Section 3.3 seems to be caused by the frequent updates required to correct the drift between different clocks. What if each node corrects its frequency instead? This creates a control system where phase error determines a frequency correction, and the integral of the frequency produces a new phase. Figure 3-11 shows a discrete-time diagram of this system.

3.4.1 Proportional Control

A proportional control strategy modifies the rate at which $\theta[n]$ progresses in proportion to the phase error:

$$\theta[n+1] - \theta[n] = \Delta\theta + K\theta_{\text{err}}[n] \qquad (3.7)$$

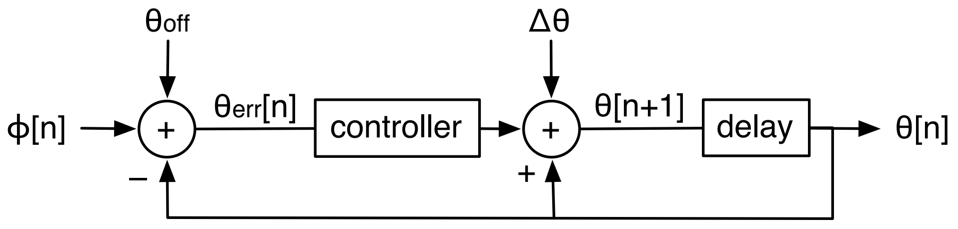

Figure 3-11: Discrete-time control system for frequency correction. The phase error is fed through the controller's transfer function and modifies the natural rate $\Delta\theta$ at which $\theta[n]$ progresses. A running sum of the rate produces a new phase value.

This same principle is used in the phase-locked loop, a common electronic circuit used to control clock signals. Note also that this update equation is equivalent to that of damped phase correction in Equation 3.2. The only difference between these two strategies is that when no new phase value is received, damped phase correction increments the phase at the node's default rate ($\Delta\theta$), while proportional control continues to increment the phase at the rate determined by the last error measurement.

The light output from a simulation, shown in Figure 3-12, demonstrates that the proportional control strategy produces steady patterns of light. The phase error reveals that the pattern is perfectly steady over time, but phase offsets between nodes still vary.

The phase error may not be perfectly stable in a real system, though. Consider variations in communication delay; the asynchronous timing between neighboring node updates may introduce unknown delays. We can approximate these effects by introducing variations in the phase values received from neighbors, as if the values were transmitted at slightly different times. Figure 3-13 shows the phase variations introduced by a $\pm 5\%$ phase noise. These variations are small compared to the average phase error at each node, so the plot shows only deviations from the average at each node. The figure shows patterns of error propagation similar to those seen with damped phase correction in Figure 3-8.

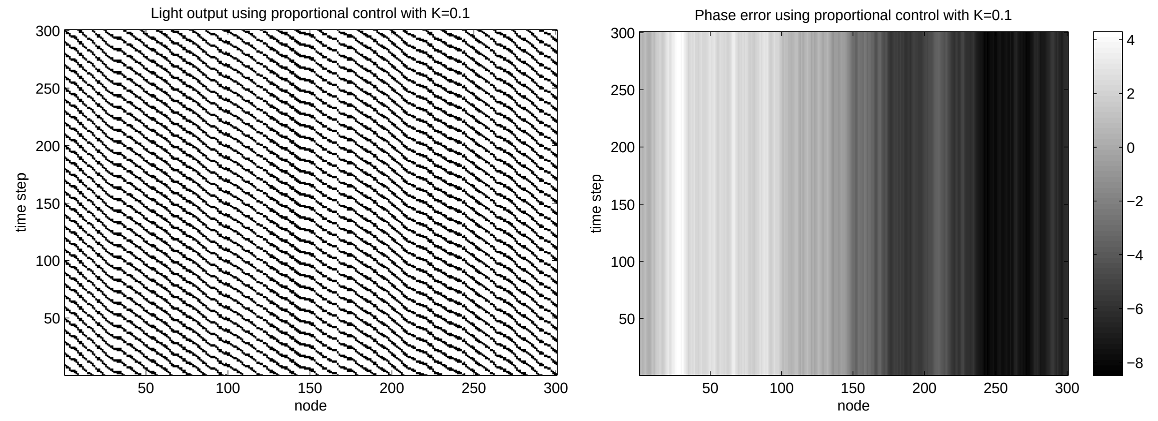

Figure 3-12: Light output and phase error using proportional control with K=0.1. The light pattern is perfectly steady over time, but shows varying offsets between nodes.

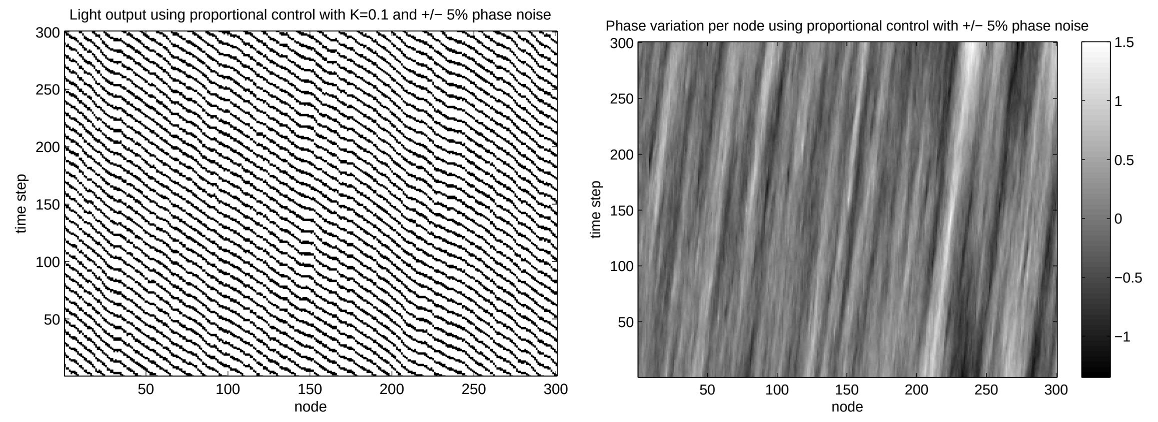

Figure 3-13: Light output (left) and difference between the phase error and mean phase error per node (right), for proportional control with $K = 0.1$ and $\pm 5\%$ phase noise added to every received value. With the much larger mean phase error removed, a pattern of error propagation is visible on the right, but the errors do not significantly affect the light output.

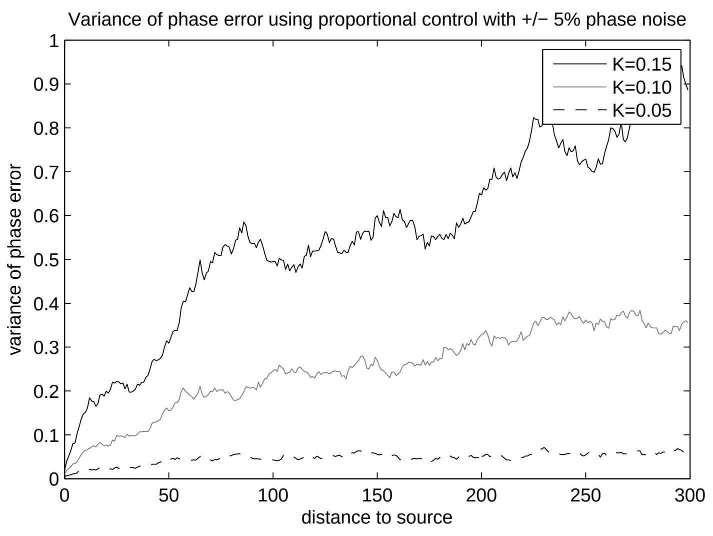

Figure 3-14: Variance of phase error using proportional control with ±5% phase noise, over 5,000 time steps. The variance increases with distance, but is smaller with greater damping.

Figure 3-14 shows that the variance of phase error increases with distance, but is small and drops quickly with a smaller gain. The simulations show that proportional control eliminates the variance caused by irregular communication, but communication delays do cause errors that propagate down the chain. These errors should not be significant if communication delays are small compared to the period of the light cycle.

Stability

With high enough gain, phase-locked loops such as this one are unstable, and can even show chaotic behavior[8]. Even in the regime where a single controller is stable, though, this system of many controllers in a chain can misbehave. Figure 3-15 shows a simulation of an unstable[3] system with $K = 0.3$.

[3]I'm using the term "unstable" loosely. If a single controller is stable, a finite chain of them will also be stable. However, a long chain may amplify noise so much that the pattern breaks down completely. The system is unstable in the sense that the response cannot be bounded as the length of the chain increases.

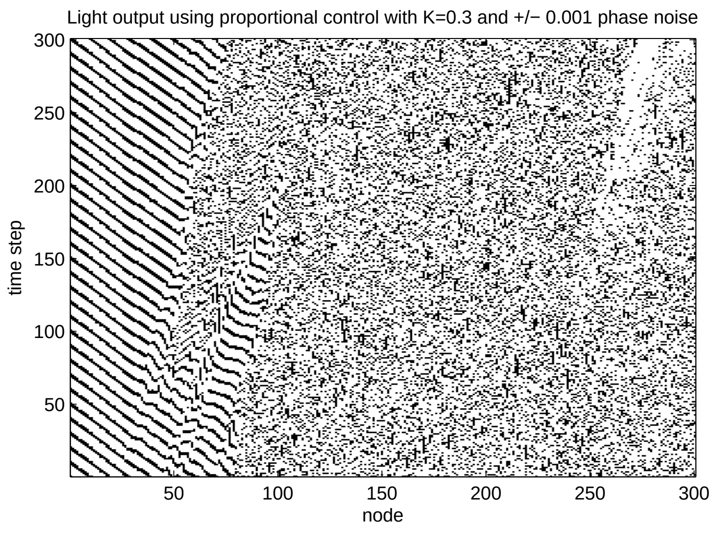

Figure 3-15: Light output using proportional control with an unstable gain of $K = 0.3$ and ± 0.001 phase noise. Gain values approaching the probability of communication (1/3) are unstable after many nodes.

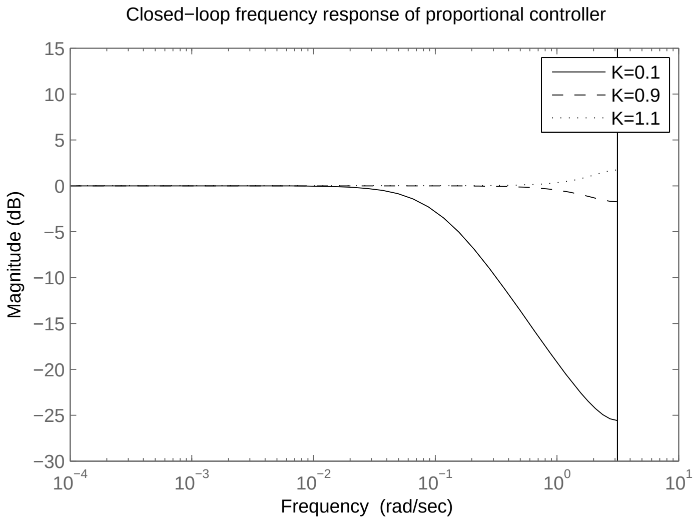

Figure 3-16: Bode magnitude plot of the closed-loop proportional controller, showing the frequency response between the received phase and the phase of the node itself, assuming perfect communication. A gain of $K > 1$ creates amplification of high frequencies near the Nyquist rate (vertical line). When many nodes are chained together, even a small amount of amplification causes instability.

To understand this instability, consider the frequency response of a node's phase to its input, shown in Figure 3-16. With perfect communication, gains of $K > 1$ create amplification of some frequencies. When many of these controllers are chained together, any noise on those frequencies gets amplified many times and causes the pattern to break down. With unreliable communication, consider a system sampled at the average update interval $1/p$. This scaled system is equivalent to a proportional controller with gain K/p, suggesting that stability requires $K/p < 1$. Thus, the system with unreliable communication should be unstable when K approaches p.

Over-damping

Like with damped phase correction, the proportional controller can be over-damped such that the lights blink out of order. Consider the steady-state condition where the controller matches the target rate $\Delta\phi$:

$$\Delta\phi = \Delta\theta + K\theta_{\text{err}}[n] \tag{3.8}$$

To make sure the nodes blink in the correct order, we need $\theta_{\text{err}}[n] < \theta_{\text{off}}$, so:

$$\begin{aligned}
\Delta\phi - \Delta\theta &= K\theta_{\text{err}}[n] \\
r\Delta\theta - \Delta\theta &= K\theta_{\text{err}}[n] \\
\Delta\theta\frac{r-1}{K} &= \theta_{\text{err}}[n] \\
\Delta\theta\frac{r-1}{K} &< \theta_{\text{off}} \\
\frac{\Delta\theta}{\theta_{\text{off}}}(r-1) &< K
\end{aligned} \tag{3.9}$$

This lower bound on K is just like that of damped phase correction, but without dependence on the communication probability p. For the parameters of the simulation, the bound shows that we should prefer $K > 0.22$.

Propagation Rate

The patterns of error in Figure 3-13 suggest that a proportional controller with $K = 0.1$ would propagate changes at a rate of one node every 10 time steps. We can define the propagation rate by looking at the *group delay* of the controller, shown in Figure 3-17. A measure used in signal processing, group delay uses phase information to compute the time delay imposed on each frequency passed through a linear system. The plot shows that a single node will delay low frequencies by up to $1/K$ samples. So, we can expect changes in the system to propagate at K nodes per time step. This is again similar to damped phase correction, without depending on the communication probability.

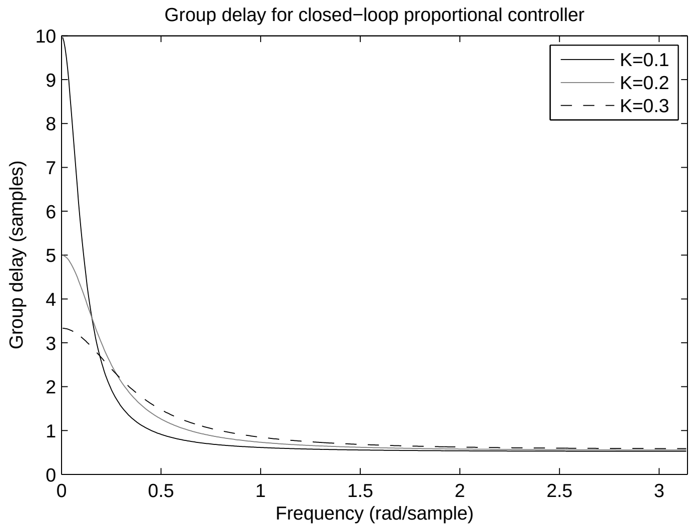

Figure 3-17: The group delay for the closed-loop proportional controller shows the amount of time that each frequency is delayed by a single node. The values for very low frequencies indicate the rate at which changes will propagate in the evacuation system.

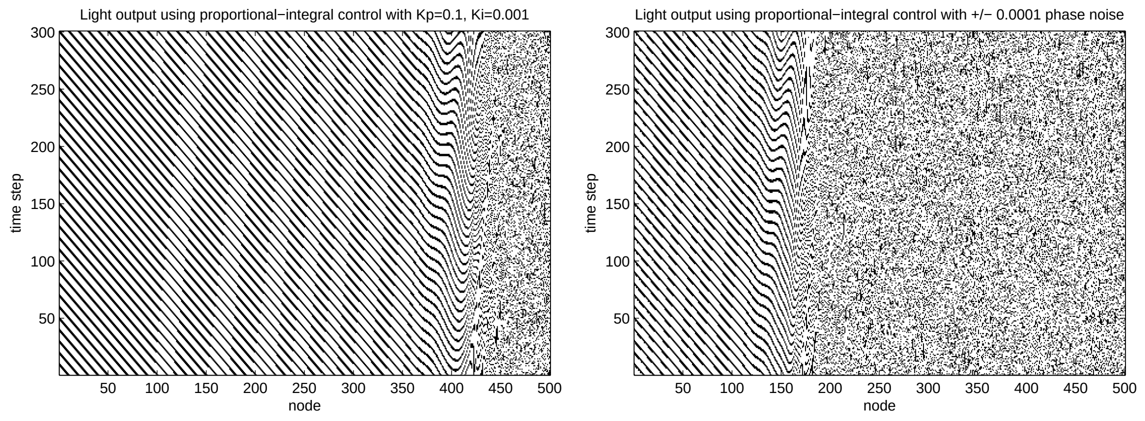

Figure 3-18: Light output using proportional-integral control with $K_p = 0.1$ and $K_i = 0.001$. On the left, the pattern devolves after 400 nodes due to the build-up of small rounding errors in the calculations. On the right, even a small amount of phase noise greatly reduces the stable region. Though the proportional-integral controller itself is stable, a long chain of them is unstable in the presence of noise.

3.4.2 Proportional-Integral Control

Proportional control produces stable patterns of light, but with varying offsets at each node. In control systems, a common strategy for removing this steady-state error is to add an integral term to the controller. In a discrete time system, this means keeping a running sum of the phase error:

$$\theta[n+1] - \theta[n] = \Delta\theta + K_p \theta_{\text{err}}[n] + K_i \sum_{t=0}^{n} \theta_{\text{err}}[t] \tag{3.10}$$

Figure 3-18 shows a simulation with a small integral constant of $K_i = 0.001$. The pattern is nearly perfect for more than 300 nodes, but suddenly devolves into chaos.

The frequency response of the controller, shown in Figure 3-19, explains this effect. With the introduction of any integral term, the frequency response shows resonance, where some frequencies are slightly amplified. When many of these controllers are chained together, any noise on those frequencies gets amplified and causes the light pattern to break down.

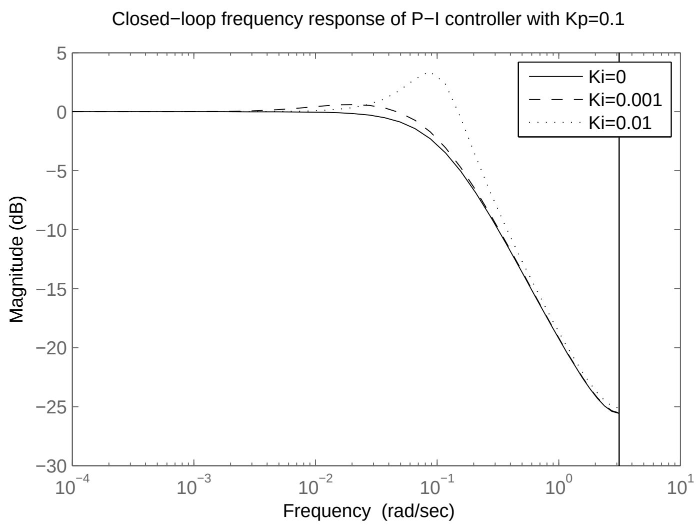

Figure 3-19: Bode magnitude plot of the closed-loop proportional-integral controller. Any integral term introduces resonance at some frequencies, where the output magnitude is larger than the input. When many of these controllers are chained together, any noise on those frequencies gets amplified and causes the light pattern to break down.

3.5 Multipath Networks

The simulations show that a proportional control strategy for phase tracking can produce a steady pattern of lights in a chain of nodes along the path to an exit. Multiple paths are not necessarily synchronized with each other, though. In an extremely dense network, such as one with nodes in every floor tile, this may be confusing.

Figure 3-20 shows a simulation of many nodes in a grid, where each row has established a stable pattern, but the rows are not synchronized with each other. One possible solution is to incorporate the phase error of neighbors on other paths. For example, neighbors with the same hop-count (according to gradient) are likely on adjacent exit paths, and should be synchronized. Consider a modified proportional control algorithm, where N is the set of neighbors with the same hop-count as the node in question:

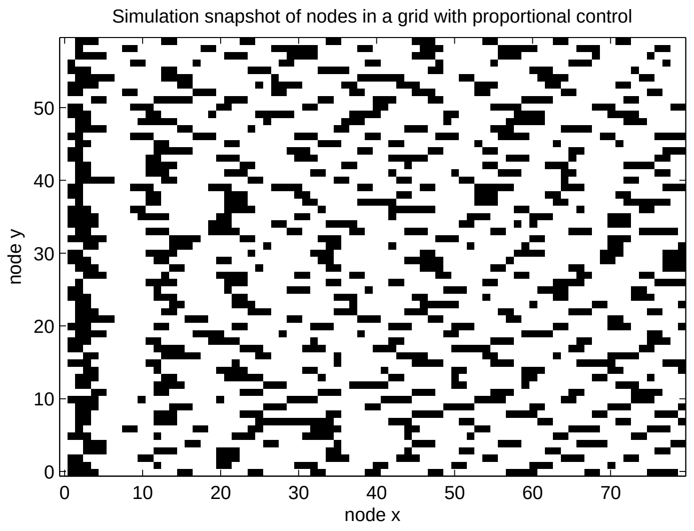

Figure 3-20: Simulation snapshot of nodes connected in a 60x80 grid, with the leftmost column of nodes as a synchronized source. Using proportional control with $K = 0.1$, each row produces a regular pattern, but the rows are not synchronized with each other.

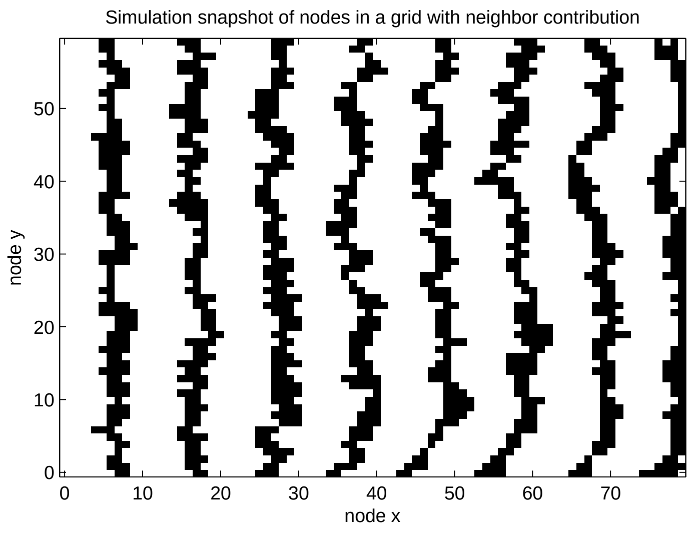

Figure 3-21: Simulation snapshot of nodes in a grid using proportional control with multiple neighbor contributions. The leftmost column is a synchronized source. Each node is affected by its error with respect to upper and lower nodes with gain $K_N = 0.02$, causing adjacent paths to synchronize.

$$\theta[n+1] - \theta[n] = \Delta\theta + K\theta_{\text{err}}[n] + K_N \sum_{i \in N} (\phi_i[n] - \theta[n]) \tag{3.11}$$

Figure 3-21 shows a simulation of this algorithm, with $K = 0.1$ and $K_N = 0.02$. Adding this small error contribution from adjacent paths causes the rows to synchronize. The nodes produce coherent waves of light sweeping towards the source.

More analysis of these complex networks is needed, but these initial simulations suggest that simple modifications to the phase tracking techniques can produce coherent patterns in a network as dense as the floor tiles of a building. The waves of light shown in Figure 3-21 would be a compelling and unambiguous evacuation guide.

3.6 Summary

This chapter presents many strategies for phase tracking, where each node along a path tracks the phase of the node ahead of it. I show that damping and frequency correction are important properties for producing steady patterns of light in large networks, despite clock variations and unreliable communication. Damping reduces the variance of phase error and smooths out discontinuities in the patterns of light, and frequency correction removes the variance caused by irregular communication intervals.

A proportional control strategy similar to a phase-locked loop has these two properties, and produces steady patterns of light in simulations of hundreds of nodes. I show that the gain should be chosen in the range $\frac{\Delta\theta}{\theta_{\text{off}}}(r-1) < K < p$ for a stable and correct pattern, and that changes in the network will propagate at approximately K nodes per time step.

I analyze the stability of these systems by considering an analogous linear control system operating on an unwrapped phase. If the closed-loop frequency response of a node shows amplification of any frequencies, then the system will be unstable with a sufficiently large network, as any noise on those frequencies will get amplified many times. For example, a proportional-integral controller amplifies some frequencies, causing the pattern to break down after many nodes.

Other strategies for the evacuation system should still be explored. For example, the firefly algorithm[13] may be a good way to keep the network synchronized. To combat long convergence times, the algorithm could perhaps be bootstrapped at start-up by a tree-based synchronization. This chapter shows, though, that phase tracking is a reasonable strategy for an evacuation system. Proportional control in particular seems effective on very large networks, and can be modified to produce synchronized patterns in dense networks with many adjacent paths.

Chapter 4

Prototype & Simulation

I verified the feasibility of the proposed evacuation system with a simple prototype and a system-level simulation. The prototype platform shows both that the proportional control algorithm works with real-world errors and variability, and that the overall system is feasible with extremely limited hardware. The system-level simulation shows that the technique of sequentially pulsing lights makes sense in the context of a building, and that the overall system achieves the goal of guiding occupants to safety, even in the face of changing threats.

4.1 Prototype Implementation

The prototype was designed to challenge the robustness of the light synchronization algorithms. It has only the necessary components to produce sequences of lights, and is missing many features that a complete system would need, but still provides a useful example to help estimate the costs and requirements of a real system.

The design of the prototype does not take into account many reliability concerns, such as power supply issues and shorts. The prototype demonstrates unreliable communication, but the only failures considered are symmetric communication failures and the complete failure of a node, as if it were removed from the system.

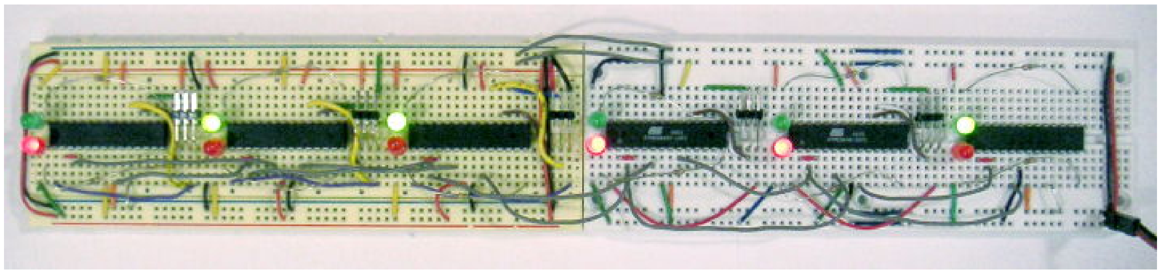

Figure 4-1: A prototype network on two breadboards. The only components for each node are an ATmega48 microcontroller and a red and green LED with resistors. Here, the six nodes are connected in a simple chain, displaying a test pattern.

4.1.1 Hardware

Figure 4-1 shows a prototype network of six nodes. Each node carries an ATmega48—a simple 8-bit microcontroller with 4KB of program memory. The self-programmable flash allows a boot-loader to be implemented, making program updates easy and simplifying algorithm development.

The prototype uses the internal RC oscillator of the ATmega48 for its clock. This choice saves the cost and component space of an external oscillator or crystal, and provides a harsh test of the algorithm's robustness. The internal oscillator is calibrated at the factory to 8 MHz ±10%.[1]

The nodes each have a red and a green LED. One LED is intended to be the primary indicator for exit paths, while the second is used for debugging.

Each node may have wired connections to at most three neighbors, limiting communication complexity and hardware costs. While fewer than three neighbors would obviously restrict the network topology, Figure 4-2 shows that three neighbor topologies might suitably occupy hallway intersections, ceiling or floor tiles, and open spaces.

To further simplify the communication system, we assume that all messages are broadcast so that any neighbor can hear them. In the prototype, a single serial output

[1]The nodes must be further calibrated within ±5% for serial communication. There is an oscillator calibration register that allows the clock rate to be calibrated to any frequency in the range 7.3-8.1 MHz within ±2% accuracy.

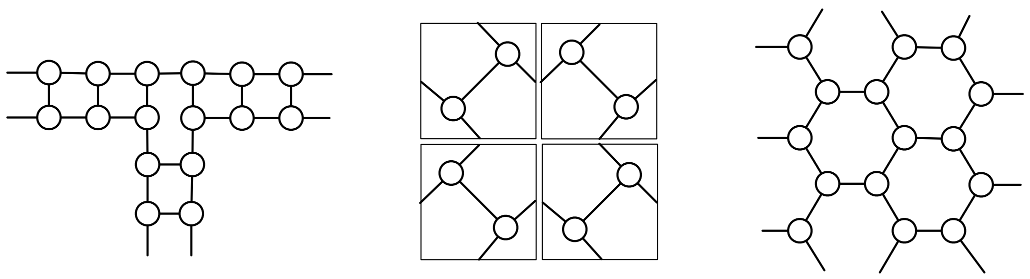

Figure 4-2: Three neighbor connection topologies. While restrictive, nodes limited to three neighbor connections can still be wired through hallway intersections (left), ceiling or floor tiles (center), and open space (right).

line is connected directly to all three neighbors. This assumption would also fit well with wireless systems, which are inherently broadcast-based, removing the need for additional communication layers.

The microcontroller code is organized into two sections: a communication system that abstracts information sharing between neighbors and manages program updates, and application code that evaluates the particular evacuation algorithm. The functions of the communication system are described below.

4.1.2 Serial Multiplexing

Like many microcontrollers, the ATmega48 includes a hardware USART (Universal Synchronous Asynchronous Receiver Transmitter) that handles the timing and bit operations necessary for RS232 serial communication, relieving the CPU of the task. For the prototype, this single serial communication line is multiplexed between the three neighbors. This allows the prototype to take advantage of the built-in hardware, but implies that only one neighbor can be heard at once, and that many communication packets may be lost.

Figure 4-3 illustrates a conceptual arrangement for multiplexing this hardware. The single output pin is wired to all three neighbors directly. The multiplexer (mux) depicted in the figure is not a hardware mux, but represents the behavior of some

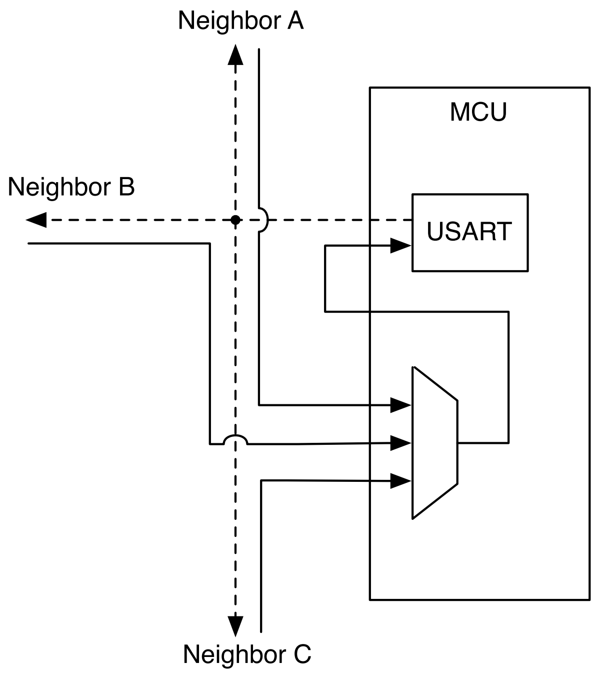

Figure 4-3: Three neighbor communication with software multiplexing. The mux represents the behavior of interrupt routines in firmware. The USART is a hardware module connected to two external pins. Input is received from the neighbors on three separate pins, and routed through an external connection to the input of the USART. The output pin of the USART (dotted line) is connected to all three neighbors.

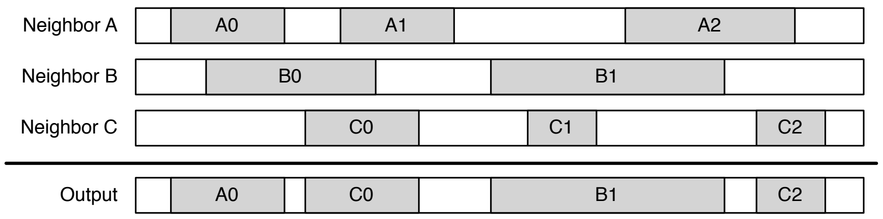

Figure 4-4: An example output of the virtual multiplexer plotted over time, given simultaneous inputs from three neighbors. During shaded regions, a message is being transmitted. The system switches to the first available transmission, but never interrupts a message or joins the middle of a one.

small interrupt routines triggered on three input pins. When the value on any of the three incoming wires changes, an interrupt is triggered that copies the change to an output pin depending on which neighbor has been selected. The output pin is externally connected back into the input pin of the USART.

Figure 4-4 shows how the firmware chooses which neighbor to listen to. It can determine whether a line is idle by checking if the time since the last change is longer than the transmission time of a byte, as the serial protocol guarantees a transition at the beginning of each byte. When no lines are busy, the first neighbor to begin a transmission gets selected. The signals are routed to the output until that line becomes idle again. Then, the next line to begin a transmission gets selected.

The interrupt routines ensure that the virtual mux does not switch away from a neighbor in the middle of a transmission, and does not switch into the middle of an ongoing transmission. As implied in the figure, some messages may be lost when neighbors transmit simultaneously, but when no messages overlap, every one can be received.

This switching scheme introduces a small problem: if two neighbors of a node are synchronized well enough that one always begins a message right before the other, then one neighbor may be consistently ignored. To prevent these repetitive collisions, each node randomly varies its broadcast interval by up to 25%.

4.1.3 Communication Abstraction

Application code is spared the details of the serial communication system. Communication is abstracted by allowing the application to share information with neighbors through special buffers in RAM: `packet_buf` and `packet_out`. Data is managed in packets of a fixed size. `packet_out` is a buffer of one packet that is regularly broadcast to neighboring nodes. `packet_buf` is a buffer of recently received packets. Application code need only read from `packet_buf` and write to `packet_out` to communicate with neighbors.

For simplicity, `packet_buf` retains a fixed number of the most recently received packets, regardless of which neighbor they came from. This enforces the idea that the algorithm not distinguish between neighbors, and approximates the idea that a neighbor is considered dead after some period of silence; traffic from other neighbors will fill the buffer until the old packets are no longer available.

4.1.4 Network Update

The communication system manages the propagation of application updates throughout the network. Each node maintains a version number, used to determine whether to broadcast or install an update. The version number is a cyclic value from 0 to 9, such that larger version numbers replace smaller ones, with the exception that version 0 may replace version 9.

Version numbers are transmitted regularly along with data packets. If a node receives an "old" version number from a neighbor, the node broadcasts a program update. If it receives an update with a "newer" version number, it installs the update and resets itself.

The high nibble of the version number also indicates whether a program should actually be executed. For example, version numbers such as 0x04 indicate incomplete programs, causing nodes to remain in boot-loader mode, whereas an update with version 0x14 will be executed immediately. Thus, large application programs can be transmitted in multiple updates, where only the last update causes the code to be executed.

Figure 4-5 shows how each type of memory in the ATmega48 is used. The application update, which encodes only changed sections of memory, is stored in non-volatile EEPROM so that it may be verified and retransmitted easily. Thus, the size of an update is limited by the EEPROM to 256 bytes. While large application programs can be sent in pieces, this architecture does not allow significant changes to the communication code.

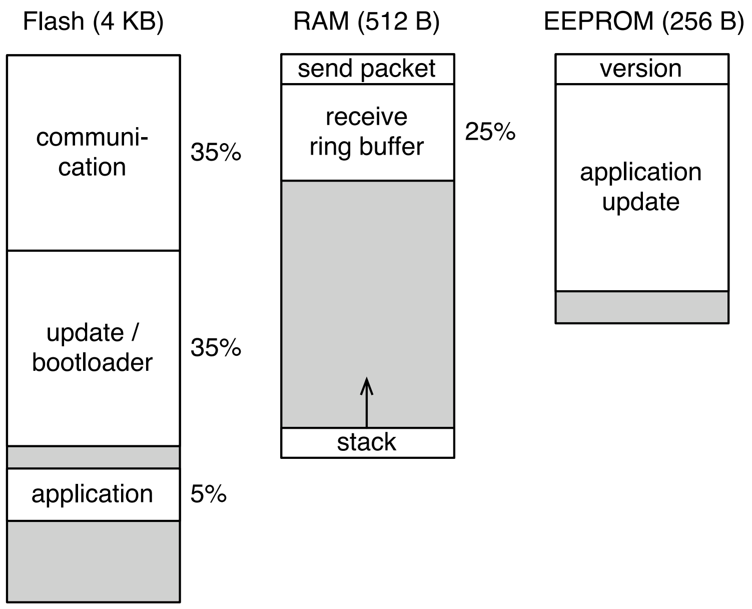

Figure 4-5: ATMega48 memory allocation (not to scale). Grey areas are unused, except for some small variables that are not shown. Note that executable code occupies most of flash. Communication code and application code share the send and receive buffers in RAM, and the boot-loader stores updates in EEPROM.

4.1.5 Data Formats

Packets are transmitted as printable ASCII characters to simplify debugging, as shown in Table 4.1. For example, each byte of a packet is encoded into two bytes representing the two digits of the value in hex. Check-sums are computed such that each data byte and the check-sum byte sum to zero (mod 256). Because of the encoding into hex characters, the check-sums provide detection of burst errors up to 16 bits, but not necessarily two separate bit errors.

4.1.6 Results

Figure 4-6 shows the light output of the prototype using undamped phase correction. Small glitches can be seen, produced by the discontinuities predicted in Section 3.3.1. Figure 4-7 shows that proportional control has very stable behavior on the prototype,

Table 4.1: Packet formats for serial communication. Data is encoded in hex characters, introducing a 2x overhead, but making the messages easily printable.

`d001122334456`	
`d . . . . . .`	type (data packet)
`00 . . . . .`	program version
`11223344 .`	data (4 bytes)
`56`	check-sum of version and data

`p0903F4:00000001FF`	
`p . . . . . . . .`	type (program update)
`09 . . . . . . .`	program version
`03 . . . . . .`	update length in bytes
`F4 . . . . .`	check-sum of version and length
`:00000001FF`	program update in Intel Hex File format

and handles transitions smoothly. When no exit is present, nodes tend to blink together, which is more desirable than the random output produced by undamped phase correction.

The proportional gain of 0.0625 was chosen as a power of two (2^{-4}) so that it can be implemented as a simple bit-shift, and is well above the lower bound suggested by the over-damping analysis in Section 3.4.1: In the prototype, node clocks were calibrated to within 5% (required for serial communication), so in the worse case the clock ratio $r = 1.05/0.95$. The phase is incremented by 2000 up to a maximum of $2^{16} - 1$, so $\Delta\theta \approx 0.06\pi$. $\theta_{\text{off}} = \pi/2$, so by Equation 3.9 the gain should be greater than 0.0128.

The analysis in Section 3.4.1 predicts that proportional control should propagate changes at $K = 0.0625$ nodes per time step. The prototype updates its phase about 26 times a second, so changes should propagate across 5 nodes in about 3 seconds. This seems to match observations in Figure 4-7 (C and D), but might be too slow for a real evacuation system. The prototype was not designed to optimize this, so there are many simple ways to improve its reaction time. The gain can be increased; higher gains still seem to be stable, but didn't seem to synchronize as well when

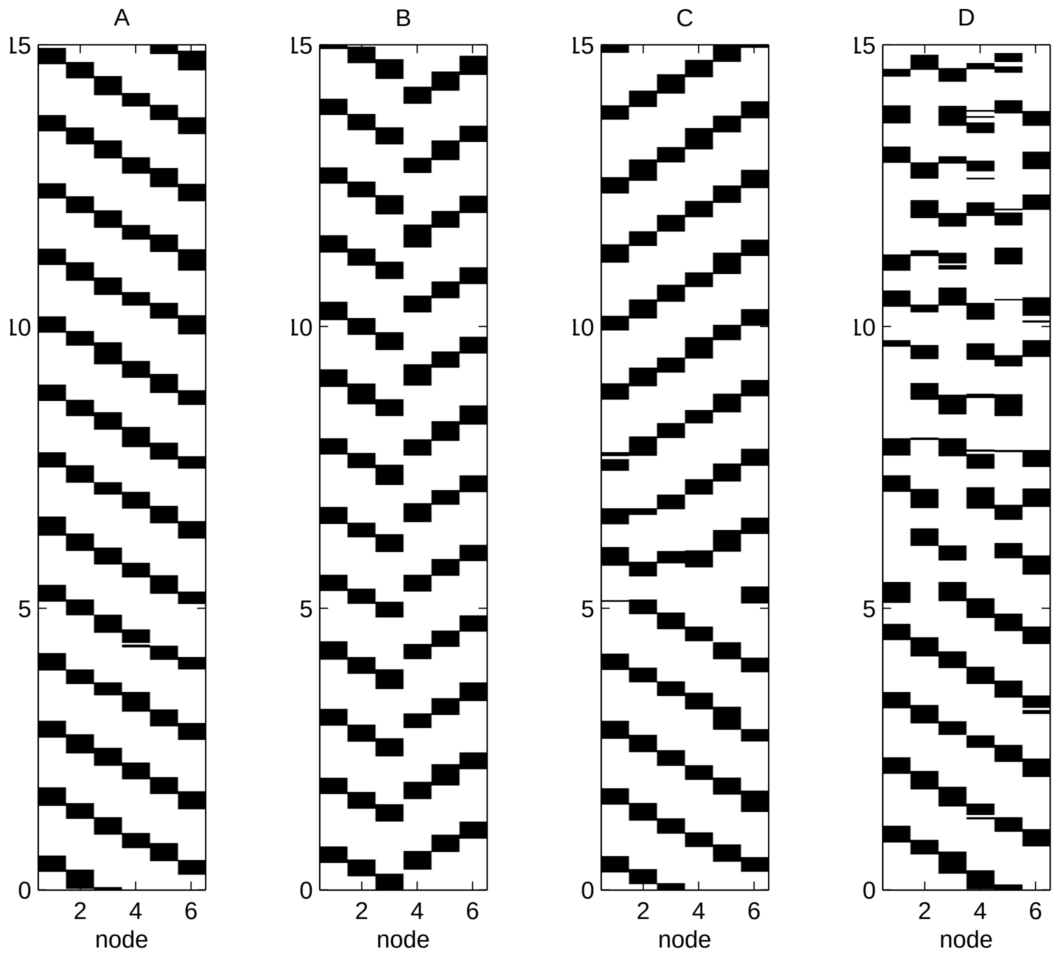

Figure 4-6: Recorded light output from prototype using undamped phase correction, with six nodes connected in a chain. (A) Steady-state behavior with node 1 marked as an exit. Note the glitch caused by a phase update near $t = 4$. (B) Nodes 1 and 6 marked as exits. Note that the two paths are not synchronized. (C) Exit marking moved from node 1 to node 6 at $t = 5$. (D) Exit mark on node 1 removed at $t = 5$. Occasional glitches can be seen again at $t = 1, 3$. Note that when the gradient is changing or no exits are present, undamped phase correction behaves unpredictably.

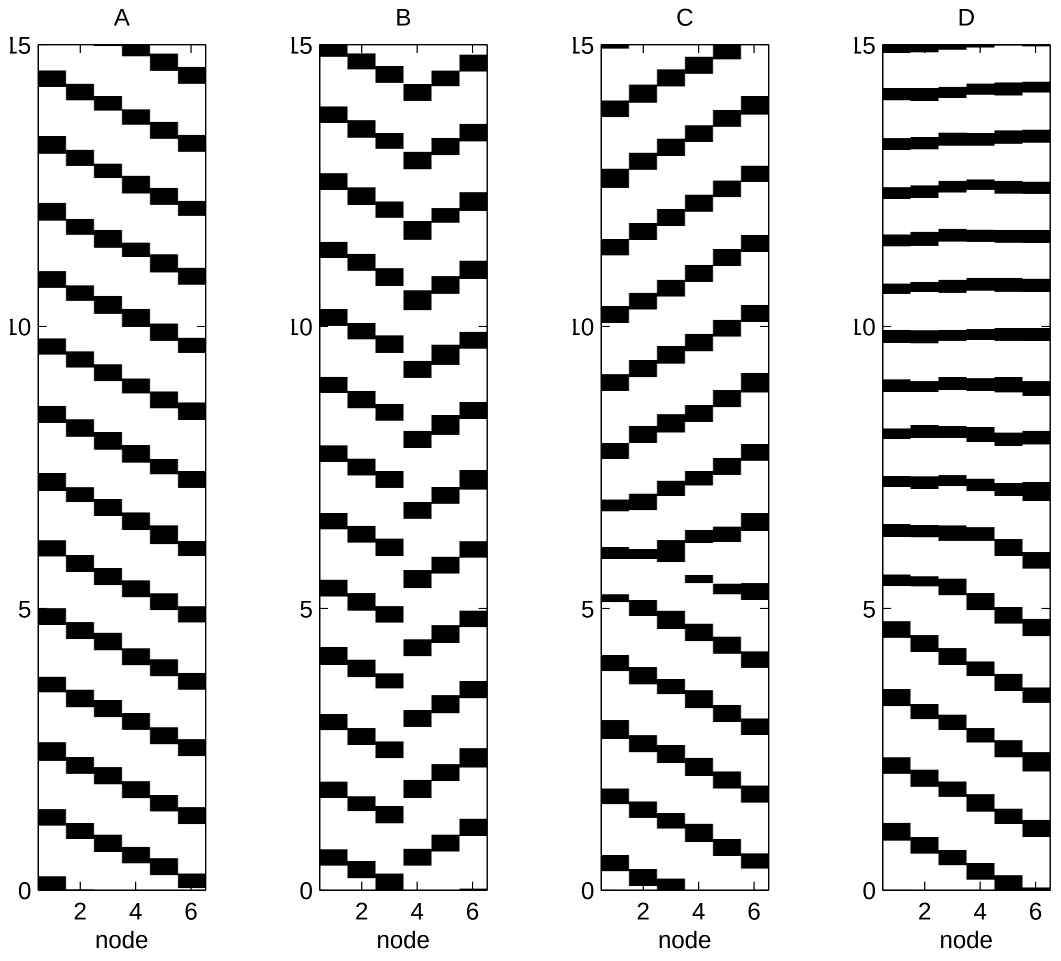

Figure 4-7: Recorded light output from prototype using proportional control with 0.0625 gain. (A) Steady-state behavior with node 1 marked as an exit. (B) Nodes 1 and 6 marked as exits. (C) Exit marking moved from node 1 to node 6 at $t = 5$. (D) Exit mark on node 1 removed at $t = 5$. Proportional control makes smooth transitions when the gradient changes, and nodes tend to blink together when no exit is present.

there are no exits. Communication delay can be improved by triggering an update as soon as a new phase is received, instead of polling a receive buffer. Communication reliability can be improved by replacing the multiplexing scheme to allow simultaneous reception. Communication baud rate can be increased and shorter messages and encodings can be used, allowing updates to be transmitted more frequently. With these improvements, the same prototype hardware should be capable of responding to changes orders of magnitude faster.

The prototype demonstrates that the major functionality of this system requires only a few inexpensive parts per node. For example, consider a system of nodes mounted on the ceiling, composed of circuit boards 1–2 inches square with surface-mount components and connected by RJ-9 cables. Figure 4.2 shows price estimates for the basic electronic components of the system. While a complete system would need circuitry for power management and protection, cables and casing, producing such small devices at commercial scales may still cost on the order of \$5 per node. Spacing nodes a couple yards apart from each other, a building installation would cost \$1.25 per square yard, or 14 cents per square foot. For comparison, a basic carpet from Home Depot costs between \$3 and \$9 per square yard (June 2008).

Table 4.2: USD. price estimates for a possible implementation at quantity 100+, from Digikey in May 2008.

ATmega48 microcontroller	\$1.50
2 LEDs	\$0.50
thermistor	\$0.10
3 RJ-9 connectors	\$1.20
resistors/capacitors	\$0.70
total basic components per node	\$4.00

4.2 System Simulation

Chapter 3.1 primarily studied the behavior of a single chain of nodes. As information only flows outward along the gradient from the exit nodes, this analysis applies to every branch in the spanning tree of a static gradient. However, a real system may have many interacting paths and a constantly changing gradient, especially when sensors are involved. The analysis and prototype behavior show that the proportional control algorithm for synchronization seems to behave smoothly with changing and conflicting gradients, but this behavior should also be studied in the context of a building.

The system simulation shown in Figure 4-8 demonstrates that the combination of a threat avoidance calculation, proportional control for synchronization, and changing threats behaves sensibly in a large building. For this example, only one floor of the building has been considered, with stairwells marked as additional exits. Threats can be added live with the mouse, and nodes or exits removed.

When threats are added, for example, the increasing gradient causes nearby nodes to blink in unison until the pattern converges to an alternate path. When all paths are unsafe, the network indicates a path through the smaller threat. The simulation seems to produce a reasonable and clear message at every point in the building, indicating that this combination of algorithms and architecture are indeed suitable for building evacuation.

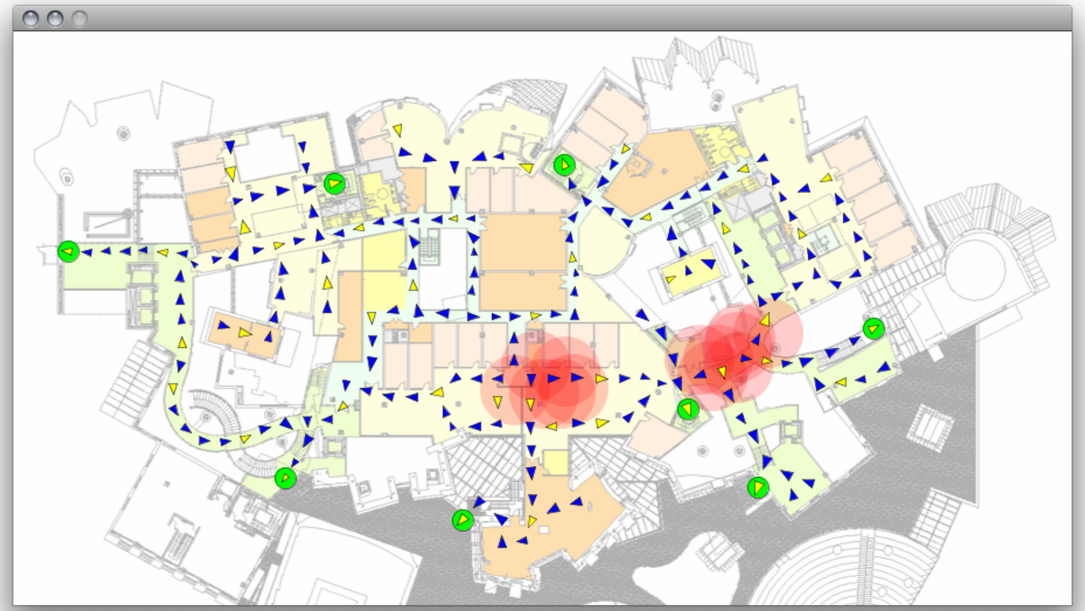

Figure 4-8: Simulation of a network of about 200 nodes installed on the third floor of the MIT Stata Center, with proportional control for light synchronization and threat avoidance based on sensor readings. Node color represents light output. While in a real system the lights would be the only indicators, the nodes are drawn here as triangles which point in the direction of the gradient. Nodes circled in green are marked as exits. Red areas represent zones of increased threat, as might be determined by temperature sensors.

Chapter 5

Contributions

In this thesis, I present and analyze a distributed system for building evacuation, where hundreds of small devices densely cover all areas of a building. The system uses simple, cheap components, and can be implemented for less than the cost of carpeting. It produces sweeping patterns of light that guide occupants to safety, and can respond to changing threats and damage to the network.

The distributed algorithm for this behavior combines a self-healing gradient with a phase-locked loop to produce sequences of lights. The analysis showed that damping and frequency correction reduce errors in large networks, and how frequency response can be used to assess the stability of many controllers connected together.

For this system to be implemented, many other issues will have to be resolved. Power should be distributed through many small back-up batteries in such a way that electrical shorts and other damage in one part of the network does not interfere with other parts. Extra functionality will be needed for triggering and controlling the alarm, and interfacing with existing fire systems. The system also needs a physical design that is unobtrusive, aesthetically pleasing, and easy to install and maintain. The algorithms explored here could be implemented on a wide variety of platforms, from wired or wireless devices mounted on walls to tiny circuits embedded in floor or ceiling tiles.

The design of this evacuation system suggests many new problems and ideas to explore. What if there are no more safe exits in a building? With a threat avoidance algorithm, perhaps nodes next to windows can be assigned a small survival probability, so that they become a target if no other exits are safe. Can the system be extended to have many source nodes marking each exit, for better robustness? What happens if an exit is blocked or jammed with people? If nodes had a way of sensing traffic, like pressure sensors in the floor, the system might be able to actively control congestion. Similar algorithms could even direct city evacuation for a natural disaster or other large emergency.

My emphasis on scalability and simple low-cost hardware is meant to touch on some larger issues as well. Sensor networks, paintable computing, amorphous computing and many related research areas explore the challenge of designing local behaviors to achieve global goals in systems of many unreliable components. This evacuation system provides a useful example of how a high-level goal can be divided into parts, and achieved through the composition of local behaviors. The algorithm analysis indicates that the behavior of such systems is difficult—but not impossible—to predict, and provides hope for a set of tools and methods that make this kind of engineering easy. This example helps to demonstrate what can be done with systems of hundreds or even millions of tiny computers. It shows not only how useful such systems could be, but how feasible and affordable they are today.

Appendix A

Prototype Source Code

A.1 comsys.h

```
// Light pattern prototype for building evacuation system
// shared definitions for communication system and application code

// Intended for ATMega48 or ATMega88 microcontroller
// Uses AVR Libc 1.4.6: http://www.nongnu.org/avr-libc/

#include <inttypes.h>

#define APP_SECTION __attribute__ ((section (".appcode")))

#define F_CPU 8000000
#define BAUD 9600

#define PACKET_BUF_SIZE 32 // power of two for easy modulo
#define PACKET_BUF_MODULO (PACKET_BUF_SIZE-1) // bitwise AND with this
#define PACKET_SIZE 4

typedef volatile uint8_t semaphore;

volatile uint8_t packet_buf_pos;
volatile uint8_t packet_buf[PACKET_BUF_SIZE][PACKET_SIZE];
volatile uint8_t packet_out[PACKET_SIZE];
//semaphore packet_buf_sem;
//semaphore packet_out_sem;

///////////// defined in comsys.c ////////////////

uint8_t db_falling_edge1(uint8_t signal);
//void wait(semaphore *s);
//void signal(semaphore *s);

////////// defined by <application>.c ///////////

void APP_SECTION run_application();
```

A.2 comsys.c

```
/*
Communication routines and bootloader

Serial input is received from 3 neighbors via
a software mux.  A pin change interrupt on the
serial lines copies values to an output pin
which should be externally connected back into
the USART.

PB6, PC0, PD4 are the serial inputs
PD1 is the serial output
PD0 and PD2 should be connected for loopback

All serial communication is ascii-readable.
Communication is broadcast-based, meaning that
no input requires a response and no response
to any output is expected.

data packets:

d001122334456
d . . . . . .  type (data packet)
 00 . . . . .  program version
   11223344 .  data (4 bytes)
           56  check-sum of version and data

program updates:

p0903F4:00000001FF
p . . . . . . . .  type (program update)
 09 . . . . . . .  program version
   03 . . . . . .  update length in bytes
     F4 . . . . .  check-sum of version and length
       :00000001FF program update in Intel Hex File format

The update length only counts record lenth, address, and data bytes.
The program update must be terminated with an end-of-file record.

The program update is stored in abbreviated form in EEPROM:
address 1 is the program version, followed by any number of lines.
a line has one byte of length, two bytes of address, then data.
a length of zero indicates the end of the update.

The first byte of EEPROM is used as the clock calibration byte,
unless it's value is 0xFF.


Timer 0 is used to check whether pins are idle
Timer 1 is used for regular packet broadcasts
*/


#include <inttypes.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/boot.h>
#include <avr/pgmspace.h>
#include <avr/wdt.h>

#include "comsys.h"
```

```
#ifdef ATmega48
#define BOOT_SECTION __attribute__ ((section (".text")))
#else
#define BOOT_SECTION __attribute__ ((section (".bootloader")))
#endif

#define EE_OSCCAL_ADDR 0
#define EE_VER_ADDR 1
#define EE_LEN_ADDR 2
#define EE_PROG_START 3
#define EEPROM_SIZE 256

////////////////////////////////////////////////// utilities

// debounce signal, returns 1 on falling edge
uint8_t db1_lowcount;
uint8_t db_falling_edge1(uint8_t signal) {
  if (signal)
    db1_lowcount = 0;
  else if (db1_lowcount < 255)
    db1_lowcount++;
  return (db1_lowcount == 100);
}

void reset() {
  // use the watchdog timer to reset the avr
  // note: must disable the watchdog timer at the beginning of main()
  wdt_enable(WDTO_15MS);
  while(1);
}

// semaphores
uint8_t trywait(semaphore *s) {
  uint8_t cSREG = SREG;
  // we don't know whether interrupts are disabled, so store SREG
  cli();
  if (*s > 0) {
    *s--;
    SREG = cSREG;
    return 1;
  } else {
    SREG = cSREG;
    return 0;
  }
}
void wait(semaphore *s) {
  while (!trywait(s));
}
void signal(semaphore *s) {
  *s++;
}

////////////////////////////////////////////////// serial/USART

void init_serial() {
  // set baud rate
  uint8_t ubrr = F_CPU/16/BAUD-1;
  UBRR0H = (uint8_t)(ubrr >> 8);
  UBRR0L = (uint8_t) ubrr;
  UCSR0B = _BV(RXCIE0) | _BV(RXEN0) | _BV(TXEN0);  // enable rx & tx
}

uint8_t USART_Receive(void) {
  loop_until_bit_is_set(UCSR0A, RXC0);
  return UDR0;
```

```
}

void USART_Transmit(uint8_t data) {
  loop_until_bit_is_set(UCSR0A, UDRE0);
  UDR0 = data;
}

// transmit_str(PSTR("hello world\n"));
void transmit_str(const char *str) {
  char c;
  while (c = pgm_read_byte(str++))
    USART_Transmit(c);
}

uint8_t hex2number(uint8_t hex) {
  if (hex >= '0' && hex <= '9')
    return hex - '0';
  if (hex >= 'A' && hex <= 'F')
    return hex - 'A' + 10;
  if (hex >= 'a' && hex <= 'f')
    return hex - 'a' + 10;
  return 0;
}

uint8_t number2hex(uint8_t num) {
  if (num < 10)
    return num + '0';
  if (num < 16)
    return num - 10 + 'A';
  return 0;
}

uint8_t receive_hex() {
  uint8_t c1 = hex2number(USART_Receive());
  uint8_t c2 = hex2number(USART_Receive());
  return (c1 << 4) + c2;
}

uint8_t transmit_hex(uint8_t val) {
  USART_Transmit(number2hex(val >> 4));
  USART_Transmit(number2hex(val & 0xF));
}

////////////////////////////////////////////////////// flash

void BOOT_SECTION write_flash_page(uint16_t page, uint8_t *buf) {
  // writes a page to flash
  // page: byte-based page address (lower 6 bits ignored)
  // buf: 64 byte array

  // based on <avr/boot.h> API example

  uint16_t i;
  uint8_t sreg;

  // Disable interrupts.
  sreg = SREG;
  cli();

  eeprom_busy_wait();

  boot_page_erase(page);
  boot_spm_busy_wait();       // Wait until the memory is erased.

  for (i=0; i<SPM_PAGESIZE; i+=2)
```

```
    {
      // Set up little-endian word.

      uint16_t w = *buf++;
      w += (*buf++) << 8;

      boot_page_fill (page + i, w);
    }

  boot_page_write(page);      // Store buffer in flash page.
  boot_spm_busy_wait();       // Wait until the memory is written.

#ifndef ATmega48
  boot_rww_enable();    // Reenable RWW-section before returning to application
#endif

  // Re-enable interrupts (if they were ever enabled).
  SREG = sreg;
}

void write_flash(uint16_t addr, uint8_t *buf, uint8_t len) {
  // writes a buffer to flash by writing the necessary pages

  uint8_t page_buf[SPM_PAGESIZE];
  uint16_t page_addr = addr & ~((uint16_t)SPM_PAGESIZE - 1); // beginning of first page
  uint16_t buf_end = addr + len; // last buffer address

  uint8_t page_i = 0; // index into page_buf
  uint16_t addr_i = page_addr; // overall index

  // if addr is not at the beginning of a page,
  // copy old flash contents into page_buf
  while (addr_i < addr) {
    page_buf[page_i] = pgm_read_byte(addr_i);
    page_i++;
    addr_i++;

  }

  // now copy from buf, writing pages to flash when necessary
  while (addr_i < buf_end) {
    page_buf[page_i] = *buf++;

    page_i++;
    addr_i++;

    if (page_i == SPM_PAGESIZE) {
      write_flash_page(page_addr, page_buf);
      page_addr = addr_i;
      page_i = 0;
    }
  }

  // if the page isn't full, copy the old flash contents into the rest
  while (page_i != 0) {
    page_buf[page_i] = pgm_read_byte(addr_i);
    page_i++;
    addr_i++;

    if (page_i == SPM_PAGESIZE) {
      write_flash_page(page_addr, page_buf);
      break;
    }
  }
}
```

```
//////////////////////////////////////////////////////// loopback mux

// output: PD2
// input:
// PB6 = PCINT6
// PC0 = PCINT8
// PD4 = PCINT20

void init_mux() {
  // set up timer 0 to check for idle neighbors
  TCCR0A = _BV(WGM01); // set auto reload
  TCCR0B = _BV(CS02); // select clk/256
  OCR0A = 3 * F_CPU/256/(BAUD/10); // time 1 bytes
  TIMSK0 = _BV(OCIE0A); // enable compare interrupt

  PCICR = _BV(PCIE0) | _BV(PCIE1) | _BV(PCIE2);
  PCMSK0 = _BV(PCINT6);
  PCMSK1 = _BV(PCINT8);
  PCMSK2 = _BV(PCINT20);
}

uint8_t listening = 0;
uint8_t idle = 0;
uint8_t available = 0;

ISR(TIMER0_COMPA_vect) {
  // any idle line should not be listening
  listening &= ~idle;
  // line is available when idle and not listening to anyone
  if (listening == 0)
    available |= idle;

  idle = 0xFF;
}

ISR(PCINT0_vect) {
  if (listening & _BV(0))
    PORTD = PINB >> 4;
  else if (available & _BV(0)) {
    PORTD = PINB >> 4;
    available = 0;
    listening = _BV(0);
  }
  idle &= ~_BV(0);
}

ISR(PCINT1_vect) {
  if (listening & _BV(1))
    PORTD = PINC << 2;
  else if (available & _BV(1)) {
    PORTD = PINC << 2;
    available = 0;
    listening = _BV(1);
  }
  idle &= ~_BV(1);
}

ISR(PCINT2_vect) {
  if (listening & _BV(2))
    PORTD = PIND >> 2;
  else if (available & _BV(2)) {
    PORTD = PIND >> 2;
    available = 0;
    listening = _BV(2);
  }
  idle &= ~_BV(2);
}


////////////////////////////////////////////////////// program update

uint8_t replaces(uint8_t new_v, uint8_t old_v) {
  // returns a boolean indicating whether a program with version new_v
  // should replace a program with version old_v

  // only look at bottom 4 bits
  new_v &= 0xF;
  old_v &= 0xF;

  return ((new_v == 0 && old_v >= 9)
          || (new_v == 9 && old_v != 9 && old_v != 0)
          || (new_v < 9 && new_v > old_v));
}

enum RECORD_TYPE {DATA=0, EOF=1};

void download_program(uint8_t version, uint8_t length) {
  // disable all interrupts except loopback
  UCSR0B = _BV(RXEN0) | _BV(TXEN0);
  TIMSK0 = 0;
  TIMSK1 = 0;
  TIMSK2 = 0;

  // only enable the loopback interrupt for the pin we are listening to
  PCICR = listening;

  PORTB = 0; // all lights on

  if (length > EEPROM_SIZE - EE_PROG_START - 3)
    reset();  // length restricted further because we don't always check for overflow

  uint8_t ee_buf[EEPROM_SIZE];
  uint8_t ee_buf_i = 0;

  enum RECORD_TYPE record_type = DATA;

  USART_Transmit('>');

  while (record_type != EOF) { // line loop

    // colon
    if (USART_Receive() != ':')
      reset();

    // length
    uint8_t len = receive_hex();
    uint8_t cksum = len;
    ee_buf[ee_buf_i++] = len;

    // address
    uint8_t data = receive_hex();
    cksum += data;
    ee_buf[ee_buf_i++] = data;
    data = receive_hex();
    cksum += data;
    ee_buf[ee_buf_i++] = data;

    // record type
    record_type = receive_hex();
    cksum += record_type;

    // data
    for (; len > 0; len--) {
      data = receive_hex();
      cksum += data;
      ee_buf[ee_buf_i++] = data;

      if (ee_buf_i > length)
        reset();
    }

    // checksum
    cksum += receive_hex();
    if (cksum != 0)
      reset();

    // end of line
    // consumes either \r\n or \n for compatibility
    data = USART_Receive();
    if (data == '\r')
      data = USART_Receive();
    if (data != '\n')
      reset();
  }

  // check length, write to eeprom
  if (ee_buf_i != length)
    reset();
  eeprom_write_byte((uint8_t *)EE_LEN_ADDR, length);

  // write the update to eeprom
  uint8_t *ee_addr = (uint8_t *)EE_PROG_START;
  uint8_t buf_i;
  for (buf_i = 0; buf_i < ee_buf_i; buf_i++)
    eeprom_write_byte(ee_addr++, ee_buf[buf_i]);

  // write the update to flash
  buf_i = 0;
  while (1) {
    uint8_t len = ee_buf[buf_i++];
    if (len == 0) break;

    uint16_t addr = (uint16_t)ee_buf[buf_i++] << 8;
    addr += ee_buf[buf_i++];

    write_flash(addr, &(ee_buf[buf_i]), len);
    buf_i += len;
  }

  // only update the version when you know the download is complete
  eeprom_write_byte((uint8_t *)EE_VER_ADDR, version);

  USART_Transmit('!');

  reset();
}

// prevents program transmission from interrupting packet transmission
semaphore transmit = 1;

void send_program() {

  if (!trywait(&transmit))
    return;

  // we don't want to be interrupted
  // note: interrupts will be re-enabled when the ISR returns
  cli();
  // this is going to take a while, so don't
  // resume listening to someone afterwards
  // (might enter the middle of a message)
  listening = 0;
  idle = 0;
  available = 0;

  uint8_t data; // temp
  uint8_t cksum;

  USART_Transmit('p');
  data = eeprom_read_byte((uint8_t *)EE_VER_ADDR);
  cksum = data;
  transmit_hex(data); // version
  data = eeprom_read_byte((uint8_t *)EE_LEN_ADDR);
  cksum += data;
  transmit_hex(data); // length
  transmit_hex(-cksum); // header checksum

  uint8_t *ee_addr = (uint8_t *)EE_PROG_START;

  while (1) {
    USART_Transmit(':');

    // length
    uint8_t len = eeprom_read_byte(ee_addr++);
    cksum = len;
    transmit_hex(len);

    // address
    data = eeprom_read_byte(ee_addr++);
    cksum += data;
    transmit_hex(data);
    data = eeprom_read_byte(ee_addr++);
    cksum += data;
    transmit_hex(data);

    // record type
    if (len == 0) {
      transmit_hex(EOF);
      cksum += EOF;
    } else {
      transmit_hex(DATA);
      cksum += DATA;
    }

    uint8_t i;
    for (i = 0; i < len; i++) {
      data = eeprom_read_byte(ee_addr++);
      cksum += data;
      transmit_hex(data);
    }

    // checksum
    transmit_hex(-cksum);

    USART_Transmit('\n');

    if (len == 0) break;
  }
```

```
  signal(&transmit);
}

////////////////////////////////////////////////// broadcast

void init_broadcast() {
  // setup timer 1 to reguarly broadcast the packet
  TCCR1B = _BV(WGM12) | _BV(CS11) | _BV(CS10); // select clk/64
  OCR1A = .1 * F_CPU / 64; // 10 times a second (limit .5 at clk/64)
  TIMSK1 = _BV(OCIE1A); // enable compare interrupt
}

uint8_t random; // used to randomly vary broadcast interval,
// to prevent constant collisions.  see serial ISR below.

ISR(TIMER1_COMPA_vect) {
  sei(); // allow recursive interrupts

  if (trywait(&transmit)) {

    USART_Transmit('d');
    uint8_t cksum = eeprom_read_byte((uint8_t *)EE_VER_ADDR);
    transmit_hex(cksum); // version

    uint8_t i;
    for (i=0; i<PACKET_SIZE; i++) {
      transmit_hex(packet_out[i]);
      cksum += packet_out[i];
    }
    transmit_hex(-cksum);
    USART_Transmit('\n');

    signal(&transmit);
  }

  random += random >> 4; // we only use low bits, so add in high nibble
  OCR1A ^= (random & 7) << 10;  // vary the broadcast interval
}


////////////////////////////////////////////////// main

int main(void) {
  // turn off the watchdog timer
  wdt_reset();
  MCUSR = 0;
  wdt_disable();

  // read oscillator calibration byte from eeprom
  uint8_t osc_cal = eeprom_read_byte((uint8_t *)EE_OSCCAL_ADDR);
  if (osc_cal != 0xFF)
    OSCCAL = osc_cal;

  DDRB = 0x3F;
  DDRD = _BV(1) | _BV(2);
  PORTB = 0xFF;
  PORTC = 0xFF;  // pull-ups
  PORTD = 0xFF;

  packet_buf_pos = 0;

  init_serial();
  init_mux();
  init_broadcast();
  sei();

  // only run the application if the version is final (0x1*)
  if (eeprom_read_byte((uint8_t *)EE_VER_ADDR) & 0x10)
    run_application();

  PORTB &= ~_BV(0); // otherwise, red light on
  PORTB |= _BV(1);
  while (1);

  return 0;
}

enum RX_STATES {CMD, PACKET_H, PACKET_L, PROGRAM_H, PROGRAM_L} rx_state;
uint8_t packet[PACKET_SIZE+1]; // includes version
uint8_t packet_pos = 0;
uint8_t packet_byte = 0;
uint8_t packet_cksum = 0;

ISR(USART_RX_vect) {
  uint8_t data = UDR0;

  sei(); // allow recursive interrupts

  random = TCNT1L + data; // take counter value and data as somewhat random bits

  uint8_t my_version;

  switch (rx_state) {
  case CMD:
    switch (data) {
    case 'd':
      rx_state = PACKET_H;
      packet_pos = 0;
      packet_cksum = 0;
      break;
    case 'p':
      rx_state = PROGRAM_H;
      packet_pos = 0;
      packet_cksum = 0;
      break;
    }
    break;

  case PACKET_H:
    packet_byte = hex2number(data) << 4;
    rx_state = PACKET_L;
    break;

  case PACKET_L:
    packet_byte += hex2number(data);
    packet_cksum += packet_byte;

    if (packet_pos == PACKET_SIZE+1) {
      if (packet_cksum == 0) {
        // check version
        my_version = eeprom_read_byte((uint8_t *)EE_VER_ADDR);
        if (replaces(my_version, packet[0]))
          send_program();

        // write packet
        packet_buf_pos = (packet_buf_pos + 1) & PACKET_BUF_MODULO;
        for (packet_pos = 0; packet_pos < PACKET_SIZE; packet_pos++)
          packet_buf[packet_buf_pos][packet_pos] = packet[packet_pos+1];
      }
```

```
          rx_state = CMD;
        } else {
          packet[packet_pos] = packet_byte;
          packet_pos++;
          rx_state = PACKET_H;
        }
        break;

      case PROGRAM_H:
        packet_byte = hex2number(data) << 4;
        rx_state = PROGRAM_L;
        break;

      case PROGRAM_L:
        packet_byte += hex2number(data);
        packet_cksum += packet_byte;

        if (packet_pos == 2) { // have received 2 bytes + cksum
          if (packet_cksum == 0) {
            // check version
            my_version = eeprom_read_byte((uint8_t *)EE_VER_ADDR);
            if (replaces(packet[0], my_version))
              download_program(packet[0], packet[1]);
          }
          rx_state = CMD;
        } else {
          packet[packet_pos] = packet_byte;
          packet_pos++;
          rx_state = PROGRAM_H;
        }
        break;
    }
}
```

A.3 evacuation.app.c

```
#include "comsys.h"
#include <avr/io.h>
#include <util/delay.h>

// undamped phase tracking and proportional control algorithms

// first two bytes of packet are the gradient value
// second two bytes are the phase

// red light (PB0) indicates whether this is an exit
// green light (PB1) is the main output

#define PROPORTIONAL  // remove definition for undamped phase tracking

#define PHASEVEL 2000     // desired amount to increment each loop
#define PHASEOFFSET 0.25  // desired difference b/t me and my neighbor
#define ON_FRACTION 0.25  // fraction of cycle that light is on
#define P_CONST_SHIFT 4  // proportional gain exponent (2^-4)

void APP_SECTION run_application() {
  uint16_t phase = 0;
  uint16_t increment = PHASEVEL;  // actual increment (adjusted)
  uint16_t prev_phase = 0; // to tell when we get a new phase

  PORTB |= _BV(0) | _BV(1); // lights off
  while (1) {
    // find packet with minimum gradient
    uint16_t min = 65535;
    uint16_t min_phase = 0;
    uint8_t buf_i = packet_buf_pos;
    uint8_t buf_end = (buf_i - 8) & PACKET_BUF_MODULO;
    // check the last 8 packets in buffer
    for (; buf_i != buf_end; buf_i = (buf_i - 1) & PACKET_BUF_MODULO) {
      uint16_t temp = packet_buf[buf_i][0] << 8;
      temp += packet_buf[buf_i][1];
      if (temp < min) {
        min = temp;
        min_phase = (uint16_t)packet_buf[buf_i][2] << 8;
        min_phase += packet_buf[buf_i][3];
      }
    }

    if (!(PINB & _BV(7))) {  // jumper says I am an exit
      min = 0;
      increment = PHASEVEL;
      PORTB &= ~_BV(0);
    } else if (min_phase != prev_phase) {

#ifdef PROPORTIONAL
      int16_t error = (min_phase + (int16_t)(PHASEOFFSET*65536)) - phase;
      // note that error must be signed to wrap around correctly
      increment = (error >> P_CONST_SHIFT) + PHASEVEL;
#else
      phase = min_phase + (int16_t)(PHASEOFFSET*65536);
#endif

      prev_phase = min_phase;
      PORTB |= _BV(0);
    }

    // populate the outgoing packet
    if (min == 65535) {
      packet_out[0] = 255;
      packet_out[1] = 255;
    } else {
      min++;
      packet_out[0] = min >> 8;
      packet_out[1] = min & 0xFF;
    }
    packet_out[2] = phase >> 8;
    packet_out[3] = phase & 0xFF;

    // main output
    if (phase < (uint16_t)(ON_FRACTION*65536))
      PORTB &= ~_BV(1);
    else
      PORTB |= _BV(1);

    phase += increment;
    _delay_loop_2(0);
  }
}
```

Bibliography

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. Amorphous computing. MIT Artificial Intelligence Laboratory memo no. 1665, August 1999.

[2] Jonathan Bachrach, Radhika Nagpal, Michael Salib, and Howard Shrobe. Experimental results and theoretical analysis of a self-organizing global coordinate system for ad hoc sensor networks. *Telecommunications Systems Journal, Special Issue on Wireless System Networks*, 2003.

[3] Jacob Beal, Jonathan Bachrach, Dan Vickery, and Mark Tobenkin. Fast self-healing gradients. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 1969–1975, New York, NY, USA, 2008. ACM.

[4] Thomas F. Burkman, Sr. and Gary Cummings. Emergency guidance system. United States Patent 4,347,499, August 1982.

[5] Michael Buschmann, Thomas Goulet, Frank Herstix, and Waldemar Ollik. Method and apparatus for marking an escape route. United States Patent 6,998,960, February 2006.

[6] William Joseph Butera. *Programming a Paintable Computer*. PhD thesis, MIT, February 2002.

[7] Adam Eames. Enabling path planning and threat avoidance with wireless sensor networks. Master's thesis, MIT, June 2005.

[8] T. Endo and L.O. Chua. Chaos from phase-locked loops. *Circuits and Systems, IEEE Transactions on*, 35(8):987–1003, August 1988.

[9] M. E. M. Lijding, H. P. Benz, N. Meratnia, and P. J. M. Havinga. Smart signs: Showing the way in smart surroundings. Technical Report TR-CTIT-06-20, University of Twente, Enschede, April 2006.

[10] Dennis Lucarelli and I-Jeng Wang. Decentralized synchronization protocols with nearest neighbor communication. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 62–68, New York, NY, USA, 2004. ACM.

[11] Mihai Marin-Perianu, Nirvana Meratnia, Maria Lijding, and Paul Havinga. Being aware in wireless sensor networks. In *15th IST Mobile & Wireless Communication Summit*, Myconos, Greece, June 2006.

[12] Clifford A. Megerle. Adaptive escape routing system. United States Patent 6,778,071, August 2004.

[13] Renato E. Mirollo and Steven H. Strogatz. Synchronization of pulse-coupled biological oscillators. *SIAM Journal on Applied Mathematics*, 50(6):1645–1662, 1990.

[14] NFPA. *NFPA 72®: National Fire Alarm Code®*. National Fire Protection Agency, 2007.

[15] Shiro Nishino. Emergency alarm and evacuation system. United States Patent 3,969,720, July 1976.

[16] David L. Reed. Premise evacuation system. United States Patent 7,154,379, December 2006.

[17] Robert Right and Hilario Costa. Fire alarm system with method of building occupant evacuation. United States Patent 7,218,238, May 2007.

[18] Fikret Sivrikaya and Bulent Yener. Time synchronization in sensor networks: A survey. In *IEEE Network*, volume 18, pages 45–50, 2004.

[19] D. Steingart, J. Wilson, A. Redfern, P.K. Wright, R. Romero, and L. Lim. Augmented cognition for fire emergency response: An iterative user study. In *11th Int. Conf. on Human-Computer Interaction (HCI)*, Las Vegas, NV, July 2005.

[20] Reed Tator. Emergency guidance system. United States Patent 6,472,994, October 2002.

[21] Jan Erik Vadseth. Guiding light system and lighting strip. United States Patent 5,815,068, June 1996.

[22] J. van Greunen and J. Rabaey. Lightweight time synchronization for sensor networks. In *2nd ACM Intl. Workshop on Wireless Sensor Networks and Applications (WSNA)*, San Diego, CA, September 2003.