Chapter 1
Composition and Scaling Challenges in Sensor Networks: An Interaction-Centric View

T. Abdelzaher

Abstract Moore's law, automation considerations, and the pervasive need for timely information lead to a next generation of distributed systems that are open, highly interconnected, and deeply embedded in the physical world by virtue of pervasive sensing and sensor-based decision-making. These systems offer new research challenges that stem from scale, composition of large numbers of components, and tight coupling between computation, communication, and distributed interaction with both physical and social contexts. These growing challenges span a large spectrum ranging from new models of computation for systems that live in physical and social spaces, to the enforcement of reliable, predictable, and timely end-to-end behavior in the face of high interactive complexity, increased uncertainty, and imperfect implementation. This chapter discusses the top challenges in composing large-scale sensing systems and conjectures on research directions of increasing interest in this realm.

1.1 Introduction

The envisioned proliferation of networked sensing devices, predicted in the 1990s, has given rise to myriads of challenges that arise from interconnecting sensors at large scale. Future distributed sensing systems will surpass the current paradigms for embedded computing, where a number of sensors and actuators implement well-understood and tightly managed control loops. New paradigms will involve data acquisition at a significantly wider scale, offering less structure and less control over the properties of the resulting loops from sensing to decision-making. It is envisioned that by the end of the next decade, a significant number (if not the majority) of Internet clients will constitute sensors and embedded devices. Indeed, the main role of future networks will shift from offering a mere communication medium between end-points to offering *information distillation* services bridging the gap between the

T. Abdelzaher (✉)
University of Illinois at Urbana-Champaign, Champaign, IL, USA
e-mail: zaher@cs.uiuc.edu

S. Nikoletseas, J.D.P. Rolim (eds.), *Theoretical Aspects of Distributed Computing in Sensor Networks,* Monographs in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-642-14849-1_1,

myriads of real-time low-level data feeds and the high-level human decision needs. The success of Google, built around the mission of organizing and "distilling" Web content, attests to the increasing use of networks as information sources. The proliferation of sensing devices gives rise to new information acquisition paradigms, such as opportunistic and participatory sensing [1, 4, 9, 15, 18], that rely on sensory data collection by individuals or devices acting on their behalf and on sharing these data at large scale to extract information of common use. New challenges arise in supporting the information distillation requirements of such emerging applications. These challenges are brought about primarily by scale and the need to compose sensing systems of large numbers of components, while maintaining predictable end-to-end properties.

Composition challenges arise from the complex interactions that large-scale sensing systems exhibit in several spaces including functional, data, and temporal interactions. This chapter focuses on four important interaction challenges that arise by virtue of scale. Namely, we elaborate on composition challenges in the face of functional interactions, data interactions, timing interactions, and interactions of system dynamics in largely distributed sensing systems.

It should be noted that this chapter is by no means a complete account of sensor network design and performance challenges. Most prominently, the chapter does not directly address the issues of heterogeneity, programming interfaces, middleware, and architectural paradigms used to facilitate building large systems. These software and architectural solutions, as well as examples of large-scale deployed networks, are detailed elsewhere in this book. The chapter also does not address the issues of security; a growing concern in recent literature as sensor networks empark on new mission-critical application domains where secure operation must be assured. The interaction challenges mentioned above present a more basic categorization of challenges, classifying them not by the software layer in which they arise (such as operating systems, communication protocols, middleware, or programming support), but rather by the conceptual space in which they occur, such as functional, temporal, or data related. These spaces are described in the following sections, respectively.

1.2 Functional Interactions

The first interaction space for components of sensor network applications is the space of *functional interactions*. Most deployed distributed system failures are attributed to unexpected interactions between multiple components that lead to new subtle failure modes. In sensor networks, the space of such interactions often cannot be fully explored at design time. Significant advances have been made in the area of formal methods and verification techniques. However, they remain limited by scalability challenges that arise due to massive concurrency and unreliable components (e.g., wireless). Most of the system development time, therefore, is spent on debugging as opposed to new component development. Debugging individual components is relatively simple. The problem lies in understanding failures that arise due to component composition in complex systems.

1.2.1 Troubleshooting Interactive Complexity

New fundamental theory, algorithms, analysis techniques, and software tools are needed to help uncover root causes of errors resulting from interactions of large numbers of components in heterogeneous networked sensing systems. System heterogeneity and tight integration between computation, communication, sensing, and control lead to a high interactive complexity. Moreover, the lack of layering and isolation, attributed to resource constraints, make it hard to apply the usual software engineering techniques aimed at reducing interactive complexity, thus further increasing the possibility of distributed errors and unexpected failures. On the other hand, software reuse is impaired by the customized nature of application code and deployment environments, making it harder to amortize debugging, troubleshooting, and tuning cost. Hence, while individual devices and subsystems may operate well in isolation, their composition may result in incompatibilities, anomalies, or failures that are very difficult to troubleshoot. At the same time, users of such systems (such as domain scientists) may not be experts on networking and system administration.

Automated techniques are needed for troubleshooting the system both at development time and after deployment in order to reduce production as well as ownership costs. Using such automated techniques, developers of future networked sensing systems should be able to significantly curtail debugging effort. Similarly, upon network deployment, service agents should be able to quickly diagnose and resolve problems in unique customer installations, hence reducing ownership cost. The aim is to answer developer or user questions such as "Why does this sensor network suffer unusually high service delays?", "Why is data throughput low despite availability of resources and service requests?", "Why does my time synchronization protocol fail to synchronize clocks when the localization protocol is run concurrently?", or "Why does this vehicle tracking system suffer increased false alarms when it is windy?[1]" Building efficient troubleshooting support to address the above questions offers significant research challenges brought about by the nature of interaction bugs, such as:

- *Non-reproducible behavior:* Interactions in networked sensing systems feature an increased level of concurrency and thus an increased level of non-determinism. In turn, non-determinism generates non-reproducible bugs that are hard to find using traditional debugging tools.
- *Non-local emergent behavior:* By definition, interaction bugs do not manifest themselves when components are tested in isolation. Current debugging tools are very good at finding bugs that can be traced to individual components. Interaction bugs manifest only at scale as a result of component composition. They result in emergent behavior that arises when a number of seemingly individually sound components are combined into a network, which makes them hard to find.

[1] In a previous deployment of a magnetometer-based wireless tracking system, wind resulted in antennae vibration which was caught by the magnetometers and interpreted as the passage of nearby ferrous objects (vehicles).

A successful approach to the automation of diagnosis of interactive complexity failures must rely on two main design principles aimed at *exploiting* concurrency, interactions, and non-determinism to *improve* the ability to diagnose problems in resource-constrained systems. These principles are as follows:

- *Exploiting non-reproducible behavior:* Exploitation of non-determinism to improve understanding of system behavior is not new to computing literature. For example, many techniques in estimation theory, concerned with estimation of system models, rely on introducing noise to explore a wider range of system states and hence arrive at more accurate models. Machine learning and data mining approaches have the same desirable property. They require examples of both good and bad system behavior to be able to classify the conditions correlated with good and bad. In particular, note that conditions that cause a problem to occur are correlated (by causality) with the resulting bad behavior. Root causes of non-reproducible bugs are thus inherently suited for discovery using data mining and machine learning approaches as the lack of reproducibility itself and the inherent system non-determinism improve the odds of occurrence of sufficiently diverse behavior examples to train the troubleshooting system to understand the relevant correlations and identify causes of problems.
- *Exploiting interactive complexity:* Interactive complexity describes a system where scale and complexity cause components to interact in unexpected ways. A failure that occurs due to such unexpected interactions is therefore not localized and is hard to "blame" on any single component. This fundamentally changes the objective of a troubleshooting tool from aiding in stepping through code (which is more suitable for finding a localized error in some line, such as an incorrect pointer reference), to aiding with diagnosing a *sequence of events* (component interactions) that lead to a failure state. For example, sequence mining algorithms present a suitable core analytic engine for diagnostic debugging.

1.2.2 Troubleshooting Examples

An example application of the above principles is reported in a previous investigation of a diagnostic debugging tool prototype. This prototype was experimented with over the course of 1 year to understand the strengths and limitations of the aforementioned approach [24–26]. The examples below give a feel for how diagnostic debugging is used according to this investigation.

1.2.2.1 A "Design" Bug

As an example of catching a design bug, we summarize a case study, published in [26], involving a multi-channel sensor network MAC-layer protocol from prior literature [27] that attempts to utilize channel diversity to improve throughput. The protocol assigned a home channel to every node, grouping nodes that communicated much into a cluster on the same channel. It allowed occasional communication between nodes in different clusters by letting senders change their channel

temporary to the home channel of a receiver to send a message. If communication failed (e.g., because home channel information became stale), senders would scan all channels looking for the receiver on a new channel and update home channel information accordingly. Testing revealed that total network throughput was sometimes worse than that of a single-channel MAC. Initially, the designer attributed it to the heavy cost of communication *across* clusters. To verify this hypothesis, the original protocol, written for MicaZ motes, was instrumented to log radio channel change events and message communication events (send, receive, acknowledge) as well as related timeouts. It was tested on a motes network. Event logs from runs where it outperformed a single-channel MAC were marked "good." Event logs from runs where it did worse were marked "bad." Discriminative sequence mining applied to the two sets of logs revealed a common pattern associated prominently with bad logs. The pattern included the events `No Ack Received`, `Retry Transmission on Channel (1)`, `Retry Transmission on Channel (2)`, `Retry Transmission on Channel (3)`, executed on a large number of nodes. This quickly led the designer to understand a much deeper problem. When a sender failed to communicate with a receiver in another cluster, it would leave its home channel and start scanning other channels causing communication addressed to it from upstream nodes in its cluster to fail as well. Those nodes would start scanning too, resulting in a cascading effect that propagated up the network until everyone was scanning and communication was entirely disrupted everywhere (both *within* and *across* clusters). The protocol had to be redesigned.

1.2.2.2 An "Accumulative Effect" Bug

Often failures or performance problems arise because of accumulative effects such as gradual memory leakage or clock overflow. While such effects evolve over a large period of time, the example summarized below [24] shows how it may be possible to use diagnostic debugging to understand the "tipping point" that causes the problem to manifest. In this case, the operators observed sudden onset of successive message loss in an implementation of directed diffusion [19], a well-known sensor network routing protocol. As before, communication was logged together with timeout and message drop events. Parts of logs coinciding with or closely preceding instances of successive message losses were labeled "bad." The rest were labeled "good." Discriminative sequence mining revealed the following sequential patterns correlated with successive message loss:

```
Message Send (timestamp = 255),     Message Send (timestamp = 0),
Message Dropped (Reason = "SameDataOlderTimeStamp").
```

The problem became apparent. A timestamp counter overflow caused subsequent messages received to be erroneously interpreted as "old" duplicates (i.e., having previously seen sequence numbers) and discarded.

1.2.2.3 A "Race Condition" Bug

Race conditions are a significant cause of failures in systems with a high degree of concurrency. A previous case study [26] demonstrated how diagnostic debugging

helped catch a bug caused by a race condition in their embedded operating system called LiteOS [5]. When the communication subsystem of an early version of LiteOS was stress-tested, some nodes would crash occasionally and nondeterministically. LiteOS allows logging system call events. Such logs were collected from (the flash memory of) nodes that crashed and nodes that did not, giving rise to "good" and "bad" data sets, respectively. Discriminative sequence mining revealed that the following sequence of events occurred in the good logs but not in the bad logs: `Packet Received`, `Get Current Radio Handle`, whereas the following occurred in the bad logs but not in the good logs: `Packet Received`, `Get Serial Send Function`. From these observations, it is clear that failure occurs when `Packet Received` is followed by `Get Serial Send Function` before `Get Current Radio Handle` is called. Indeed, the latter prepares the application for receiving a new packet. At high data rates, another communication event may occur before the application is prepared, causing a node crash. The race condition was eliminated by proper synchronization.

1.2.2.4 An "Assumptions Mismatch" Bug

In systems that interact with the physical world, problems may occur when the assumptions made in protocol design regarding the physical environment do not match physical reality. In this case study [25], a distributed target tracking protocol, implemented on motes, occasionally generated spurious targets. The protocol required that nodes sensing a target form a group and elect a group leader who assigned a new ID to the target. Subsequently, the leader passed the target ID on in a leader handoff operation as the target moved. Communication logs were taken from runs where spurious targets were observed ("bad" logs) and runs where they were not ("good" logs). Discriminative pattern mining revealed the absence of member-to-leader messages in the bad logs. This suggested that a singleton group was present (i.e., the leader was the only group member). Indeed, it turned out that the leader hand off protocol was not designed to deal with a singleton group because the developer assumed that a target would always be sensed by multiple sensors (an assumption on physical sensing range) and hence a group would always have more than one member. The protocol had to be realigned with physical reality.

While these preliminary results are encouraging, significant challenges remain. For example, non-trivial scalability enhancements are needed. Scalability limitations imply that the designer should have some intuition into which event types and which event attributes to monitor. Since it is hard to tell which event types are relevant to a bug in advance, ideally, we would like to be able to monitor a large number of different event types and attributes, leaving it to a software tool to ignore irrelevant ones automatically, without degrading efficiency in identifying culprit event sequences. The patterns suspected of causing failures often have a false dependence on workload. For example, the frequency of occurrence of many events depends on the communication rate. If communication rates vary from one experiment to the next, discriminative sequence mining often zooms-in on differences in event traces caused by differences in communication rates and not by bugs. Some

form of normalization is needed. Often one or a small number of initial (bad) events create a cascading wave, where a larger number of repercussions follow, in turn setting off an even larger number of measurable consequences (manifestations of anomalous behavior). Identifying this causal *chain* (or, in many cases, *tree*) is hard due to the difference in the frequency of occurrence of events at different levels in the tree. Unsynchronized clocks often result in finding incorrect sequences making it hard to infer distributed patterns correctly. Existence of multiple bugs often causes decreased diagnostic accuracy. Events with multiple attributes (e.g., function calls with multiple parameters) cause problems and have to be broken up into series of events with single attributes, which in turn generates false event sequences. Addressing such challenges may require interdisciplinary collaboration between data mining experts, machine learning experts, sensor networks experts, and troubleshooting experts in order to provide solutions that both consider peculiarities and requirements of sensor network troubleshooting and leverage algorithmic expertise needed for root cause diagnosis.

1.3 Data Interactions

Another important interaction space for sensor networks is the space of *data interactions*. An emerging application model for networked sensing systems is that of *social sensing* [1, 4], which broadly refers to applications that employ sensors used by individuals in their homes, cars, offices, or on their person, whose measurements may be shared for purposes of various application-related services. Social sensing systems range from medical devices that measure human biometrics and share them with medical repositories available to care-givers [29], location sensors, and accelerometers in phones and cars that can be used to compute aggregate information of community interest such as pollution or traffic patterns [9, 11, 17]. Traditional embedded and networked sensing systems research typically considers computing systems that interact with physical and engineering artifacts and assumes a single trust domain. Interaction of future embedded sensing devices with both physical and social spaces (in multiple trust domains) creates new challenges, such as loss of privacy, that can be broadly classified as *data interaction* challenges.

1.3.1 Privacy and Data Aggregation

Protection mechanisms from involuntary physical exposure are needed to enforce *physical privacy*. Innovative optimization problems can be formulated by recognizing that privacy is not a binary concept. When data are continuous and noisy, privacy is more akin to a degree of uncertainty; a concept closely related to noise and filtering in control applications. Control of voluntary information sharing must facilitate privacy-preserving exchange of time-series data. A predominant use of data in social sensing applications is for aggregation purposes such as mapping distributed phenomena or computing statistical trends. New mathematically based data

perturbation and anonymization schemes are needed to hide user data but allow fusion operations on perturbed or partial data to return correct results to a high degree of approximation. These problems are difficult due to interactions between data streams.

When sharing a single data item or stream, it is easy to reason about what information is releaved and what information is withheld. When multiple streams are shared, however (possibly by different individuals), correlations between these streams may be exploited to make additional inferences that make it harder to control what exactly is being revealed. For example, sharing an acoustic energy signature from the neighborhood of a person on a campaign to reduce noise pollution may reveal something about the person's location if the noise level correlates with a train or bus schedule on a known route. This data interaction challenge makes it especially hard to reason about privacy in a social sensing system. Nevertheless, certain privacy assurances are often needed to encourage people to share the information needed for the social sensing application to function. Hence, an important challenge becomes one of understanding how to perturb (or decorrelate) data in such a way that it becomes impossible to make additional privacy-violating inferences, beyond what is explicitly allowed by the data owner.

Data aggregation operations are most common in social sensing systems where multiple data streams need to be combined to compute some community-wide information such as energy consumption trends, driving patterns, or fitness and weight loss statistics. The challenge is to perturb a user's sequence of data values such that (i) the individual data items and their trend (i.e., their changes with time) cannot be estimated without large error, whereas (ii) the distribution of the data aggregation results at any point in time is estimated with high accuracy. Intuitively, privacy in this context refers to the degree of uncertainty or error regarding a user's individual data. For instance, in a health-and-fitness application, it may be desired to find the average weight loss trend of those on a particular diet or exercise routine as well as the distribution of weight loss as a function of time on the diet. This is to be accomplished without being able to reconstruct any individual's weight and weight trend without significant error.

1.3.2 Perturbation Examples and Time-Series Data

Examples of data perturbation techniques can be found in [2, 3, 10]. The general idea is to add random noise with a known distribution to the user's data, after which a reconstruction algorithm is used to estimate the distribution of the original data. Early approaches relied on adding independent random noise. These approaches were shown to be inadequate. For example, a special technique based on random matrix theory has been proposed in [23] to recover the user data with high accuracy. Later approaches considered hiding individual data values collected from different private parties, taking into account that data from different individuals may be correlated [16]. However, they do not make assumptions on the model describing

the *evolution* of data values from a given party over time, which can be used to jeopardize privacy of data streams. By developing a perturbation technique that specifically considers the data evolution model, one can prevent attacks that extract regularities in correlated data such as spectral filtering [23] and principal component analysis (PCA) [16].

In other work [11], it was shown that privacy of time-series data can be preserved if the noise used to perturb the data is itself generated from a process that approximately models the measured phenomenon. For instance, in the weight watchers example, we may have an intuitive feel for the time scales and ranges of weight evolution when humans gain or lose weight. Hence, a noise model can be constructed that exports realistic-looking parameters for both the direction and time constant of weight changes. We can think of this noise as the (possibly scaled) output of a *virtual user*. Once the noise model is available, its structure and probability distributions of all parameters are agreed upon among all parties contributing to the aggregation result. By choosing random values for these noise parameters from the specified distribution, it is possible to generate arbitrary weight curves (of virtual people) showing weight gain or loss. A real user can then add their true weight curve to that of one or several locally generated virtual users obtained from the noise model. The actual model parameters used to generate the noise are kept private. The resulting perturbed stream is shared with the pool where it can be aggregated with that of others in the community. Since the distributions of noise model parameters are statistically known, it is possible to estimate the sum, average, and distribution of added noise (of the entire community) as a function of time. Subtracting that known average noise time series from the sum of perturbed community curves will thus yield the true community trend. The distribution of community data at a given time can similarly be estimated (using deconvolution methods) since the distribution of noise (i.e., data from virtual users) is known. The estimate improves with community size.

An important question relates to the issue of trust. Given that non-expert users cannot be asked to derive good noise models for their private data, how does a non-expert client know that a given noise model is privacy-preserving? Obtaining the noise model from an external party is risky. If the party is malicious, it could send a "bad" model that is, say, a constant, or a very fast-changing function (that can be easily separated from real data using a low-pass filter), or perhaps a function with a very small range that perturbs real data by only a negligible amount. Such noise models will not perturb data in a way that protects privacy.

Consider an information aggregation service that announces a suggested noise model structure and parameter distribution to the community of users over which aggregation is performed. The model announced by the server can be trusted by a user only if that user's own data could have been generated from some parameter instantiation of that model with a non-trivial probability. This can be tested locally by a curve fitting tool on the user's side. Informally, a noise model structure and parameter distributions are accepted by a user only if (i) the curve fitting error for user's own data is not too large and (ii) the identified model parameter values for

user's data (that result from curve fitting) are not too improbable (given the probability distributions of model parameters). A friendly user interface can be developed to automate the verification of the noise model on the user's behalf.

More formally, consider a particular application where an aggregation server collects data from a community to perform statistics. In previous work [11], a perturbation algorithm is described for a community of N individuals who share M data points with the aggregation each (we assume this to be the same across users for notational simplicity). Let $x^i = \left(x_1^i, x_2^i, \ldots, x_M^i\right)$, $n^i = \left(n_1^i, n_2^i, \ldots, n_M^i\right)$, and $y^i = \left(y_1^i, y_2^i, \ldots, y_M^i\right)$ represent the data stream, noise, and perturbed data shared by user i, respectively. At time instant k, let $f_k(x)$ be the empirical community distribution, $f_k^e(x)$ be the exact community distribution, $f_k(n)$ be the noise distribution, and $f_k(y)$ be the perturbed community distribution. Most user data streams can be generated according to either linear or non-linear discrete models. In general, a model can be written as a discrete function of index k, which can be time, distance, or other (depending on the application), parameters $\underline{\boldsymbol{\theta}}$ and inputs $\underline{\mathbf{u}}$, and is denoted as $g(k, \underline{\boldsymbol{\theta}}, \underline{\mathbf{u}})$. Notice that $\underline{\boldsymbol{\theta}}$ is a fixed length parameters vector characterizing the model while $\underline{\mathbf{u}}$ is a vector of length M characterizing the input to the model at each instance. Given the data $x^i = \left(x_1^i, x_2^i, \ldots, x_M^i\right)$, the model $g(k, \underline{\boldsymbol{\theta}}, \underline{\mathbf{u}})$, and the approximated distributions $f_\theta^n(\underline{\boldsymbol{\theta}})$, $f_u^n(\underline{\mathbf{u}})$, the perturbed data for user i is generated by (i) generating samples $\underline{\boldsymbol{\theta}}_{\mathbf{n}}^{\mathbf{i}}$ and $\underline{\mathbf{u}}_{\mathbf{n}}^{\mathbf{i}}$, from the distributions $f_\theta^n(\underline{\boldsymbol{\theta}})$ and $f_u^n(\underline{\mathbf{u}})$, respectively, (ii) generating noise stream $n^i = \left(n_1^i, n_2^i, \ldots, n_M^i\right)$, where $n_k^i = g\left(k, \underline{\boldsymbol{\theta}}_{\mathbf{n}}^{\mathbf{i}}, \underline{\mathbf{u}}_{\mathbf{n}}^{\mathbf{i}}\right)$, and (iii) generating perturbed data by adding the noise stream to the data stream $y^i = x^i + n^i$.

Now, consider reconstructing the *distribution* of community data at a given point in time. At time instance k, the perturbed data of each user are the sum of the actual data and the noise $y_k^i = x_k^i + n_k^i$. Thus, the distribution of the perturbed data $f_k(y)$ is the *convolution* of the community distribution $f_k(x)$ and the noise distribution $f_k(n)$, $f_k(y) = f_k(n) * f_k(x)$. All the distributions above can be discretized and the equation can be rewritten as: $f_k(y) = H f_k(x)$, where H is a Toeplitz cyclic matrix, which is also called the blurring kernel, constructed from the elements of the discrete distribution $f_k(n)$, $f_k(x)$ is the community distribution at time k that needs to be estimated, H is known, and $f_k(y)$ is the empirical perturbed data distribution. This problem is well known in the literature as the *deconvolution problem*. The Tikhonov–Miller restoration method [34] can be employed to compute the community distribution. It requires an a priori bound $\boldsymbol{\varepsilon}$ for the L^2 norm of the noise, together with an a priori bound M for the L^2 norm of the community distribution, $\left|\left|H f_k^e(x) - f_k(y)\right|\right|_2 \leq \boldsymbol{\varepsilon}$ and $\left|\left|(H^T H)^{-\nu} f_k^e(x)\right|\right|_2 \leq M$.

The approach preserves individual user privacy while allowing accurate reconstruction of community statistics. A multi-dimensional extension of this approach was presented in recent literature [32]. In this example, perturbation was added to the GPS trajectories of individual vehicles. The perturbed trajectories were then shared with a central server, whose responsibility was to reconstruct traffic statistics in a city. While the individuals who shared their data were allowed to "lie" about their (i) GPS location, (ii) velocity, and (iii) time of day, the reconstruction was

shown to be accurate in that it reported the true average speed on city streets as a function of correct actual location and time. Figure 1.1 (reproduced from [32]) depicts the manner in which GPS trajectories are perturbed in this approach. Figure 1.2 (reproduced from [32]) compares the distribution of ground truth speed data to the reconstructed distribution obtained from perturbed shared data. It can be seen that the two distributions are rather similar. As a measure of similarity, Table 1.1 (reproduced from [32]) compares the percentages of speeding vehicles on four city streets, obtained using original and perturbed data, respectively. It can be seen that the estimates obtained from perturbed data are reasonably accurate.

Several research questions remain. For example, what is a good upper bound on the reconstruction error of the data aggregation result as a function of the noise statistics introduced to perturb the individual inputs? What are noise generation techniques that minimize the former error (to achieve accurate aggregation results) while maximizing the noise (for privacy)? How to ensure that data of individual data streams cannot be inferred from the perturbed signal? Intuitively, this is doable because traditional filtering methods such as PCA and spectral filtering work based on the assumptions that additive noise is time independent, independent of the signal, and has a small variance compared to the signal variance. With a good perturbation scheme, these assumptions are violated. What are some bounds on minimum error in reconstruction of individual data streams? What are noise generation techniques that maximize such error for privacy? Privacy challenges further include the investigation of attack models involving corrupt noise models (e.g., ones that attempt to deceive non-expert users into using perturbation techniques that do not achieve adequate privacy protection), malicious clients (e.g., ones that do not follow

Table 1.1 Percentage of speeding vehicles

Street	Real %	Reconstructed %
University Ave	15.60	17.89
Neil Street	21.43	23.67
Washington Street	0.5	0.15
Elm Street	6.95	8.6

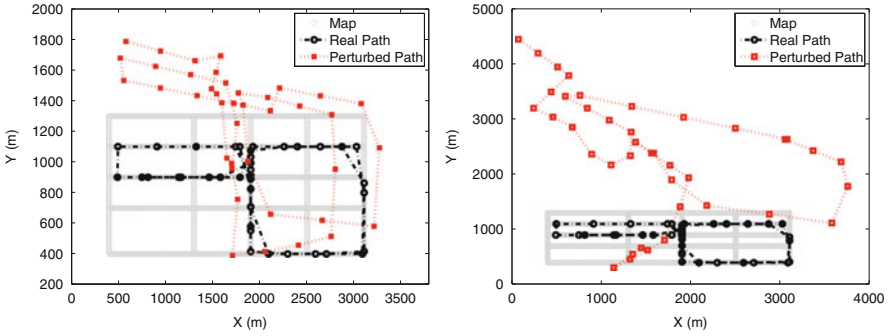

Fig. 1.1 Real and perturbed traffic trajectories for different perturbation levels

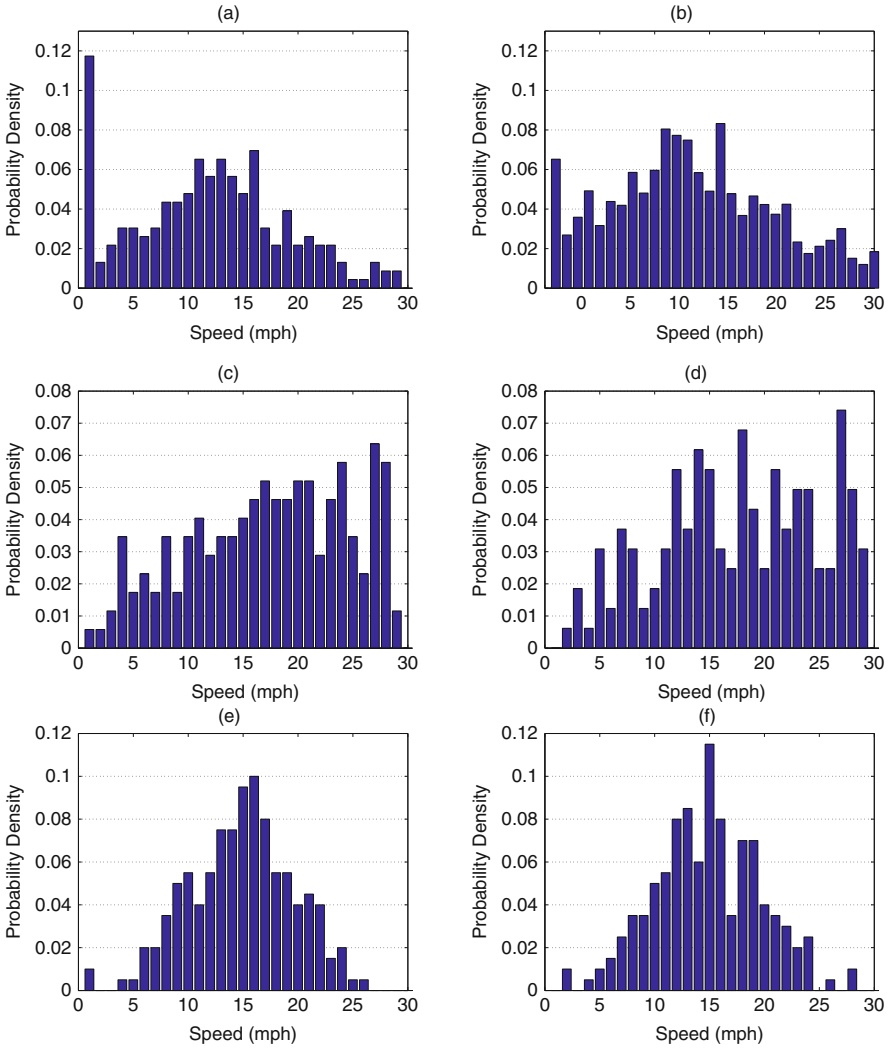

Fig. 1.2 Real and reconstructed traffic speed distributions. (**a**) Real community speed distribution; (**b**) Reconstructed speed distribution; (**c**) Real speed distribution on University Ave.; (**d**) Reconstructed speed distribution on University Ave.; (**e**) Real speed distribution on Washington Ave.; and (**f**) Reconstructed speed distribution on Washington Ave.

the correct perturbation schemes or send bogus data), and repeated server queries (e.g., to infer additional information about evolution of client data from incremental differences in query responses). For example, given that it is fundamentally impossible to tell if a user is sharing a properly perturbed version of their real weight or just some random value, what fractions of malicious users can be accommodated without significantly affecting reconstruction accuracy of community statistics? Can damage imposed by a single user be bounded using outlier detection techniques

that exclude obviously malicious users? How does the accuracy of outlier detection depend on the scale of allowable perturbation? In general, how to quantify the trade-off between privacy and robustness to malicious user data? How tolerant is the perturbation scheme to collusion among users that aims to bias community statistics? Importantly, how does the *time-series* nature of data affect answers to the above questions compared to previous solutions to similar problems in other contexts (e.g., in relational databases)? The above challenges offer significant research opportunities in the area of data interactions and social sensing.

The above perturbation techniques, defense solutions, and bounds are especially challenging due to the presence of multiple correlated data streams, or data streams with related context. For example, consider a social sensing application where users share vehicular GPS data to compute traffic speed statistics in a city. In this case, in order to compute the statistics correctly as a function of time and location, each vehicle's speed must be shared together with its current GPS location and time of day. Perturbing the speed alone does not help privacy if the correct location of the user must be revealed at all times. What is needed is a perturbation and reconstruction technique that allows a user to "lie" about their speed, location, and time of day, altogether, in a manner that makes it impossible to reconstruct their true values, yet allow an aggregation service to average out the added multi-dimensional noise and accurately map the true aggregate traffic speed as a function of actual time and space. This problem is related to the more general concern of privacy-preserving classification [36, 37], except that it is applied to the challenging case of aggregates of time-series data. Understanding the relation between multi-dimensional error bounds on reconstruction accuracy and bounds on privacy, together with optimal perturbation algorithms in the sense of minimizing the former while maximizing the latter, remains an open research problem.

1.4 Temporal Interactions

The third important interaction space for computing applications that interface with the physical world is the space of *temporal interactions*. Embedded computations must generally obey not only functional integrity constraints but also timeliness constraints on results. Early applications of sensor networks focused on soft domains where it was not critical to analyze timing properties. Eventually, as the range of sensor network applications extends to include mission-criticial and safety-critical ones, the timeliness of interactions with the physical world will become important. Recent sensor network literature reflects increasing interest in analysis of real-time behavior and time constraints [6, 13, 28, 33]. Hence, theory is needed to analyze end-to-end delay in large networked sensing systems that execute a set of distributed real-time tasks.

Timing correctness requirements arise due to data volatility, interaction with mobile objects, and the need for timely reaction to environmental events. The time-sensitive nature of sensor network applications and environmental interactions motivates understanding of the real-time limitations of information transfer, such that

future networks could be properly sized for the desired real-time transfer capability. Real-time information transfer is characterized by deadlines on data communication. In a hard real-time application, only those bits that are transferred prior to their deadlines contribute useful information. Missed deadlines result in adverse consequences that range from utility loss to significant physical damage. Deadlines could arise for various reasons, for example, the necessity to react to external events in a timely manner, and the need to deliver dynamically changing data prior to the expiration of their respective validity intervals.

Recently, information-theoretic bounds were derived for sensor networks that quantify the ability of the network to transfer bytes across distance [12, 30]. For time-sensitive applications, a more useful bound should also be a function of delay. Observe that network delay and throughput are interrelated. Intuitively, the network should be able to transfer more bits by their deadlines if the deadlines are more relaxed. Finding a function that bounds achievable total capacity subject to delay constraints is a new objective that has not been addressed in sensor network literature. We call it *real-time capacity* of distributed sensing systems.

1.4.1 Temporal Analysis of Distributed Systems

As a step toward a fundamental understanding of real-time capacity, significant advances were independently made in the embedded systems and networking communities to quantify the timing properties of distributed computation or communication. Existing techniques for analyzing delay/throughput trade-offs in distributed systems can be broadly called *decomposition based*. Decomposition-based techniques break the system into multiple subsystems, analyze each subsystem independently, then combine the results. Network calculus [7, 8] (developed in the networking community) and holistic analysis [31, 35] (developed in the embedded systems community) fall into this category.

A deficiency with decomposition-based approaches is that by viewing the aggregate problem as a set of smaller subproblems, often interactions between the subproblems are ignored or simplified. For example, network calculus does not accurately account for the effects of pipelining between stages when multiple flows share the same set of successive hops. It merely computes a delay bound on each hop based on its service curve and its arrival curve (computed from the service curve on the previous hop). Pipelining, however, makes it impossible for the same flow to suffer the delay bound on several successive hops in a row. Intuitively, this is because if a packet waits for many other packets to get processed on one hop, by the time its turn comes, most of the other packets have already moved sufficiently downstream that they may not interfere with it again. Hence, this packet's interference at the downstream hop is lower than what the bound predicts. This is an example of a temporal interaction between stages in a distributed system that leads to a *subadditive property* of individual (i.e., per hop) worst-case delay bounds. Generally speaking, the worst-case delay of a task on two successive stages of processing is *deterministically less* than the sum of its worst-case delays on the individual stages.

An exception is the top priority task that suffers no interference on its path. There is a need to quantify these subadditive delay properties of distributed computation. At present, very little work exists toward a general theory for delay composition and the relation of delay composition results to priorities of tasks and the amount of present interference.

Consider, by contrast, the fundamental laws of circuit theory used to analyze linear electric circuits. In that theory, a small number of fundamental rules (e.g., Kirchhoff laws) allow a designer to analyze complex circuits of arbitrary interconnection topology, reducing them to their effective transfer functions and deducing their exact end-to-end characteristics, such as total impedance, current draw, and voltage drop. The same compositionality is observed in feedback control theory, where component models, represented by block diagrams, can be collapsed into an equivalent single block that accurately expresses the overall system model and enables controller design. A similar theory is needed for networked sensing systems that develop rules for composition of temporal behavior of real-time system components. We call this category of techniques for analyzing distributed systems *reduction-based* (as opposed to decomposition-based) techniques. It is key for reductions to capture the essential properties of components involved and the properties of their interactions. This ensures that reductions do not lead to inaccuracies caused by ignored or oversimplified dependencies.

1.4.2 Reduction-Based Analysis and Delay Composition Algebra

A recent reduction-based approach to composition of timing properties of distributed sensing systems is delay composition algebra [21]. Given a graph of system resources, where nodes represent processing resources and arcs represent the direction of job flow, algebraic operators systematically "merge" resource nodes, composing their workloads per rules of the algebra, until only one node remains. The workload of that node represents a single resource job set called the *uniprocessor job set*. Uniprocessor schedulability analysis can then be used to determine the schedulability of the set.

Workload of any one node (that may represent a single resource or the result of reducing an entire subsystem) is described generically by a two-dimensional matrix stating the worst-case delay that each job, J_i, imposes on each other job, J_k, in the subsystem the node represents. Let us call it the *load matrix* of the subsystem in question. Observe that on a node that represents a single resource j, any job J_i that is of higher priority than job J_k can delay the latter by at most J_i's worst-case computation time, $C_{i,j}$, on that resource. This allows one to trivially produce the load matrix for a single resource given job computation times, $C_{i,j}$, on that resource. Element (i, k) of the load matrix for resource j, denoted $q^j_{i,k}$ (or just $q_{i,k}$ for notational simplicity where no ambiguity arises), is simply equal to $C_{i,j}$ as long as J_i is of (equal or) higher priority than J_k. It is zero otherwise.

The main question becomes, in a distributed system, how to compute the worst-case delay that a job imposes on another when the two meet on more than one

resource? The answer decides how delay components of two load matrices are combined when the resource nodes corresponding to these matrices are merged using appropriate algebraic operators. Intuitions derived from single resource systems suggest that delays are combined *additively*. This is not true in distributed systems. In particular, it was shown in [20] that worst-case delays in pipelines are *subadditive* because of gains due to parallelism caused by pipelining. More specifically, the worst-case delay imposed by a higher priority job, J_i, on a lower priority job, J_k, when both traverse the same set of stages varies with the *maximum* of J_i's per-stage computation times, not their *sum* (plus another component we shall mention shortly).

The delay composition algebra leverages the aforementioned result. Neighboring nodes in the resource DAG present an instance of pipelining, in that jobs that complete execution at one node move on to execute at the next. Hence, when these neighboring nodes are combined, the delay components, $q_{i,k}$, in their load matrices are composed by a maximization operation. In delay composition algebra, this is done by the PIPE operator. It reduces two neighboring nodes to one and combines the corresponding elements, $q_{i,k}$, of their respective load matrices by taking the maximum of each pair. For this reason, we call $q_{i,k}$ the *max term*.

It could be, however, that two jobs travel together in a pipelined fashion for a few stages (which we call a pipeline segment), then split and later merge again for several more stages (i.e., another pipeline segment). Consider a higher priority job J_i and a lower priority job, J_k. In this case, the max terms of each of the pipeline segments (computed by the maximization operator) must be added up to compute the total delay that J_i imposes on J_k. It is convenient to use a running counter or "accumulator" for such addition. Whenever the jobs are pipelined together, delays are composed by maximization (kept in the max term) as discussed above. Every time J_i splits away from J_k, signaling the termination of one pipeline segment, the max term (i.e., the delay imposed by J_i on J_k in that segment) is added to the accumulator. Let the accumulator be denoted by $r_{i,k}$. Hence, $r_{i,k}$ represents the total delay imposed by J_i on a lower priority job J_k over all past pipeline segments they shared. Observe that jobs can split apart only at those nodes in the DAG that have more than one outgoing arc. Hence, in the algebra, a SPLIT operator is used when a node in the DAG has more than one outgoing arc. SPLIT updates the respective accumulator variables, $r_{i,k}$, of all those jobs J_k, where J_k and a higher priority job J_i part on different arcs. The update simply adds $q_{i,k}$ to $r_{i,k}$ and resets $q_{i,k}$ to zero.

In summary, in a distributed system, it is useful to represent the delay that one job J_i imposes on another J_k as the sum of two components $q_{i,k}$ and $r_{i,k}$. The $q_{i,k}$ term is updated upon PIPEs using the maximization operator (the max term). The $r_{i,k}$ is the accumulator term. The $q_{i,k}$ is added to the $r_{i,k}$ (and reset) upon SPLITs, when J_i splits from the path of J_k. PIPE and SPLIT are thus the main operators of the algebra. In the final resulting matrix, the $q_{i,k}$ and $r_{i,k}$ components are added to yield the total delay that each job imposes on another in the entire system.

The final matrix is indistinguishable from one that represents a uniprocessor task set. In particular, each *column k* in the final matrix denotes a uniprocessor job set of jobs that delay J_k. In this column, each non-zero element determines

the computation time of one such job J_i. Since the transformation is agnostic to periodicity, in the case of periodic tasks, J_i and J_k simply represent the parameters of the corresponding periodic task invocations. Hence, for any task, T_k, in the original distributed system, the final matrix yields a uniprocessor task set (in column k), from which the schedulability of task T_k can be analyzed using uniprocessor schedulability analysis.

Finally, the above discussion omitted the fact that the results in [20] also specified a component of pipeline delay that grows with the number of stages traversed by a job and is independent of the number of higher priority jobs, called the *stage-additive* component, s_k. Hence, the load matrix, in fact, has an extra row to represent this component. As the name suggests, when two nodes are merged, this component is combined by addition. A detailed account of delay composition algebra, including a complete exact specification of its operators and examples of its use, can be found in [20–22]. One can easily envision examples from the sensor networks domain, where aggregation trees, for instance, lead to traffic patterns where transmission of individual flows (represented as a pipeline of forwarding stages) forms a DAG or convergecast graph, whose end-to-end delay may need to be bounded. Other examples include query processing applications, where a single query may be divided across multiple nodes to be evaluated against different subsets of data, then the results combined. Given multiple queries of different query processing graphs, their end-to-end timing behavior can be analyzed using the above approach.

Delay composition algebra is a step toward understanding temporal interactions and composition of timing properties in distributed sensing systems. In turn, this understanding can lead to a quantification of new notions of real-time capacity. Several questions must be answered for a useful real-time capacity theory to emerge:

- *Load metrics:* Real-time capacity must be expressed in appropriate load metrics. For example, classical schedulability bounds are expressed in terms of utilization. For distributed systems, one must determine which of the family of viable load metrics is the "best" metric to use to quantify the ability of the system to meet timing constraints.
- *Sufficient capacity regions:* Real-time capacity quantifies system load that can be supported within time constraints, which known as the *schedulability* problem. Schedulability, however, is an NP-hard problem and gives rise to very complex (porcupine) scheduable state spaces. To derive practical analytic capacity expressions, sufficient schedulability conditions must be found, meaning those defined by simple surfaces that encompass most (but not necessarily all) states in which timing constraints are met. We call them capacity regions. There is an inherent trade-off between the simplicity of capacity regions and their degree of approximation. Good compromises must be sought that maintain simplicity without introducing excessive pessimism.
- *Composition rules:* Rules must be defined for composing capacity regions of large systems from those of their subsystems. In general, starting with capacity regions of elementary components, one should be able to compose capacity regions of arbitrarily large systems. Most importantly, capacity expressions should not become more pessimistic with composition. For very large systems,

where the number of components can be viewed as infinite, continuous forms of composition rules are needed. Composition becomes an integration operation over functions of component densities as opposed to an operation that is carried out on individual components. Delay composition algebra is a first step toward defining such composition rules.

- *Optimization algorithms:* Capacity regions define sets of system states that meet sufficient time constraints. It may be desired to optimize various metrics within those constraints. For example, one might want to derive points of maximum sensor network throughput or minimum total energy consumption within capacity region boundaries.

A new real-time capacity theory should make it possible to understand the timing behavior of large real-time sensing networks with in-network computation at intermediate hops. It should also become possible to quantify end-to-end behavior of complex distributed sensing applications such as distributed power grid control and telepresence. The theory should help understand how prioritization affects real-time capacity. It will be possible, for example, to do a cost/benefit analysis of prioritizing different sensor data queues in a complex distributed sensing application since the theory will quantify the effect of prioritization on the load/timeliness trade-off. The needed theory is different from previous foundations for analysis of network delay that consider networks as graphs of links that carry packets in that the role of computation on network nodes must be considered together with communication. The real-time capacity is a general notion that does not make limiting assumptions on the types of processing resources involved. Hence, both network transmissions and CPU processing should be analyzable within the same framework. This framework is needed to understand the end-to-end timing behavior of large systems involving tightly intertwined computation, communication, and sensing.

1.5 Interactions of System Dynamics

The final interaction space for distributed sensing systems addressed in this chapter is the space of system dynamics. Control engineers are trained to analyze dynamics of physical and embedded systems and verify their adherence to desired specifications. Unfortunately, dynamics (in a control-theoretic sense) are not a term that computer scientists normally come in contact with in their education. As a result, dynamics of feedback loops that pervade the design of computing software are often poorly accounted for and poorly understood. While it is easy to use simple heuristics to ensure the stability of feedback loops within smaller subsystems, unexpected consequences may arise when such subsystems are combined.

1.5.1 Sources of Dynamics in Software

Informally, software dynamics occur when systems involve “delayed” or “cumulative” response that may be approximated by differential or difference equation

models. An example of software features that generate dynamics is an action taken by one system component that depends on results of another action taken by another component in the past. Another example is when some action or system parameter depends on a previously accumulated value of another parameter. For example, the number of packets waiting in a network queue depends on the accumulation (i.e., integral) of differences between past enqueue and dequeue rates. This dependency on previous values can be captured by a difference equation creating software dynamics.

If all causal responses within a software system were instantaneous, the system becomes strictly reactive in that it instantly reaches a state that is a function of only currently applied stimuli. However, in most systems, effects depend not only on current state but also on previous states. This is especially true in sensor networks, where queues and other communication delays create significant dependencies on past states. Hence, analysis of dynamics is needed. This analysis is especially critical in computing systems when feedback is used. Dynamics imply that software decisions are made based on *past* information (e.g., due to delays in acquiring or communicating the information) or that effects of actions are not immediately observed (for example, a reduction in source sending rates in a congested network will take some time before it diffuses network delays). If software feedback loops do not properly address dynamics, they may "under-" or "over-react." For example, sending rates might not be decreased enough to eliminate congestion, or conversely might be cut too much, thereby unnecessarily degrading performance. *Stability* is the property of a feedback loop that allows it to converge over time to desired performance. Control theory allows designers to analyze stability, convergence rate, overshoot, and other dynamic response properties of computing systems. In particular, control theory explains that while individual components may be stable, their composition may not be necessarily so. Hence, using ad hoc techniques in designing feedback in software systems may result in components that work well in isolation, but have poor performance when combined.

The above discussion suggests that composing, analyzing, controlling, and optimizing performance of large-scale networked sensing systems is an important problem, complicated by increased system size, a growing number of tunable parameters (and hence feedback loops that tune them), subtle interactions among distributed components, and limited observability of internal software state at run-time.

The problem is of growing importance. The increasing cost of managing large systems suggests that sensor networks and the information processing systems they serve will operate with progressively less human oversight. The trend toward increasingly autonomous, larger, and more interconnected systems exacerbates the problem in two important ways:

- First, autonomy implies increasing need for *adaptive* or *self-tuning* behavior. Many aspects of system functionality will be automated, creating a large number of feedback loops. For example, MAC-layer algorithms may automatically determine the best line transmission rates such that reliability is maximized. Routing may automatically determine the least-cost routes as load on different network components changes. Congestion control may automatically determine the best

application sending rate to prevent bottlenecks and overload. Application knobs such as fidelity of information processing may be manipulated depending on factors such as currently available energy or user demand.

- Second, increased system scale implies interactions among a larger number of components, which makes component composition a growing problem. As developer teams that build software systems grow, each developer becomes responsible for a progressively smaller fraction of the system, essentially leading to myopic design. Unintended interactions among different feedback loops in such a design can lead to unexpected effects on aggregate performance. Individually designed adaptive or automated modules with efficient performance management policies (when considered in isolation) might contribute to significant performance degradation when put together. Research and management tools are needed to address these performance composability problems, especially when designers and operators do not have the analytic background to analyze overall dynamics and stability of their systems.

1.5.2 Examples of Dynamic Interactions

To give an example of adverse interactions and illustrate the importance of addressing composability of dynamic behavior in the context of distributed sensing systems, consider a scenario drawn from the domain of communication protocols. Let shortest path routing be one policy that constantly discovers shorter routes between sources and destinations. Let the MAC-layer rate adaptation policy, on the other hand, tune the radio transmission rate to match channel quality (a lower rate is used on lower quality channels). While each policy is individually well motivated, composing the two policies leads to an adverse interaction. Shortest path routing may prefer longer hops (so there are fewer of them on the path). Longer hops tend to have lower quality, which causes the radio to lower its transmission rate. At the lower rate, new more distant neighbors may be discovered leading to shorter routes. Switching to those routes reduces channel quality again, leading to further rate reductions. This adverse feedback cycle ultimately diminishes throughput. Such composition problems are expected to increase in software systems as these systems become more complex (i.e., made of more components) and feature more capabilities for adaptation.

Interestingly, adverse interactions may result even when the different adaptive policies have the same objective. These unintended interactions stem from subtle incompatibilities between their performance management mechanisms. Consider a distributed data processing back-end that performs multistage data fusion for a large sensing system. Two mechanisms are installed to save energy during off-peak load conditions. The first mechanism is to power off those machines that are underutilized and distribute their load across other machines in their tier. When all machines exhibit high utilization, extra machines are powered on. We call it the *On/Off policy*. The second mechanism is to employ dynamic voltage scaling (DVS) on individual processors such that the speed and voltage of a machine are reduced when the

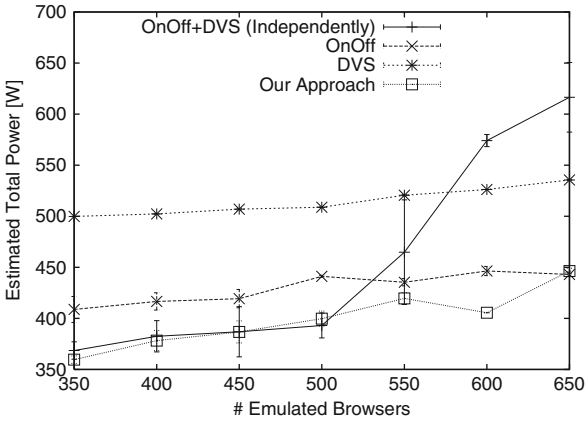

Fig. 1.3 Two adaptation policies in a multi-tier server farm and their combination

machine is underutilized and increased when it is overloaded. We call it the *DVS policy*. The two policies work well in isolation. Previous literature reports [14] that the two policies combined may actually result in a *higher* energy consumption than when one policy is used in isolation. This effect is shown in Fig. 1.3.

The explanation lies in unmodeled dynamics. If the DVS policy is aggressive enough, whenever the utilization of a machine decreases, the policy reduces clock frequency (and voltage) thus slowing down the machine and restoring a high utilization value. From the perspective of the (DVS-oblivious) On/Off policy, the farm becomes "fully utilized," as the measured utilization of all machines is high. This drives the On/Off policy to needlessly turn machines on in an attempt to relieve the full utilization condition. DVS will slow down the clock further, causing more machines to be turned on, and so on. Figure 1.3 also shows that proper joint control of both knobs (labeled "our approach" in the figure) does improve performance over tuning either knob in isolation.

To uncover unintended loops, a formal analysis of the system should use stability notions from control theory. A simplified analysis technique, based on the notion of adaptation graphs, was presented in previous computing literature [14]. Nodes in an adaptation graph represent the key variables in the system such as delay, throughput, utilization, length of different queues, and settings of different policy knobs. Arcs represent the direction of causality. For example, consider a back-end data server that serves queries over a network. When the utilization, U, of the outgoing link increases, the delay, D, of served requests increases as well (because they wait longer to be sent over the congested link). Hence, an arc exists from utilization to delay, $U \rightarrow D$, indicating that changes in the former affect the latter. The arcs in the adaptation graph are annotated by either a "+" or a "−" sign depending on whether the changes are in the same direction or not. In the example at hand, since an increase in utilization causes a *same-direction* change in delay (i.e., also an increase), the arc is annotated with a "+" sign: $U \rightarrow^{+} D$. Some of the arcs represent fundamental natural phenomena (for example, an increase in delay is a natural consequence of an increase in utilization). Others represent *programmed* behavior or

policies. For example, an admission controller may be programmed to decrease the fraction of admitted requests, R, in response to an increase in delay, D. Hence, an arc exists in the adaptation graph from delay to admitted requests, $D \rightarrow^{-} R$. The arc is annotated with a "−" sign because an increase in delay results in a change in the opposite direction (i.e., a decrease) in admitted requests. This arc does not represent a natural phenomenon but rather the way the admission control policy is programmed. These arcs are called *policy arcs* and annotated with the name of the module implementing the corresponding policy. Hence, we have $D \rightarrow^{-}_{\text{AC}} R$, where AC stands for the admission control module. Figure 1.4a depicts the adaptation graph of the data server under consideration. The graph is composed of three arcs. The arc $D \rightarrow^{-}_{\text{AC}} R$ reflects that the admission controller reduces the number of admitted requests when delay increases and vice versa. The arc $R \rightarrow^{+} U$ reflects the natural phenomenon that any changes in the number of admitted requests result in same-direction changes in outgoing link utilization. Finally, the arc $U \rightarrow^{+} D$ expresses the fact that changes in link utilization cause changes in delay (in the same direction). The three arcs form a cycle (a feedback loop). An interesting property of the loop is that the product of the signs of the arcs is negative. This indicates a negative feedback loop, which is expected for stability.

As another example, consider a network power management middleware that measures links utilization, U. If the link utilization is low, the server workload must be low. The middleware thus engages dynamic voltage scaling (DVS) on the server to lower processor voltage, V, and frequency, F, hence reducing power consumption, P, due to the off-peak load condition. This adaptation action can be expressed as $U \rightarrow^{+}_{\text{PM}} V$ and $U \rightarrow^{+}_{\text{PM}} F$, where PM stands for power management middleware (i.e., a decrease in link utilization causes the policy to decrease both voltage and frequency which explains the signs on the arcs). In turn, we have $V \rightarrow^{+} P$ and $F \rightarrow^{+} P$, which says how power consumption changes with voltage and frequency. Finally, we have $F \rightarrow^{-} D$, since lowering frequency (i.e., slowing down a processor) increases delay and vice versa. Figure 1.4b depicts the adaptation graph for the network power management middleware.

As might be inferred from above, each component or subsystem of a larger system has its own adaptation graph that describes what performance variables this component is affecting and what causality chains (or loops) exist within. When a system is composed, the adaptation graphs of individual components are coalesced. Figure 1.4c shows the combined adaptation graph that results when a server described in Fig. 1.4a operates on top of the middleware described in Fig. 1.4b. To check for incompatibilities (adverse interactions), the graph is searched for loops using any common graph traversal algorithm. Loops that traverse component boundaries are emergent behavior loops that have not been created by design. In particular, if the product of signs on one such loop is positive, the cycle indicates an unsafe feedback loop. In other words, a stimulus reinforces itself causing more change in the same direction. In control-theoretic terms, such a loop is *unstable*.

For example, in Fig. 1.4c, the cycle $U \rightarrow^{+}_{\text{PM}} F$, $F \rightarrow^{-} D$, $D \rightarrow^{-}_{\text{AC}} R$, $R \rightarrow^{+} U$ crosses module boundaries and has positive sign product, indicating that it is unstable. The cycle is an instance of an adverse interaction explained as

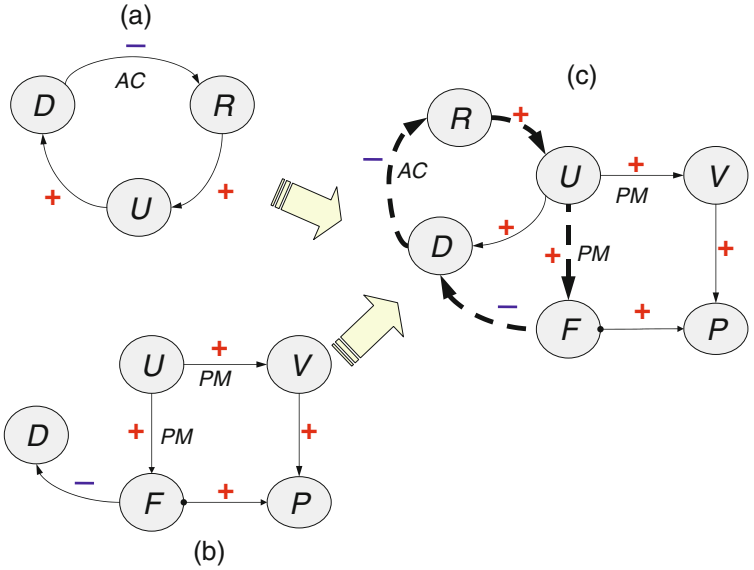

Fig. 1.4 Examples of adaptation graphs and their combination. (**a**) Adaptation graph of an admission controller of a performance-aware server; (**b**) Adaptation graph of a network power management middleware; and (**c**) Combined adaptation graph of the two

follows. Starting with the node labeled, U, when the network utilization decreases in the server, the power management middleware causes the server to slow down. This, in turn, increases the delay experienced by served requests causing the admission controller to accept fewer requests. The reduced accepted number of requests will further decrease the load on the network link, causing the power management middleware to slow down processors even more. This, in turn, may cause a more significant reduction in admitted requests and a further reduction in network load. This cycle could ultimately bring the server to a crawl, indeed an adverse consequence of unintended interaction.

Analytic foundations and tools are needed for the design, composition, and optimization of performance of large-scale distributed, adaptive, sensing systems. Much of our future infrastructure, such as power grids, homeland defense systems, and disaster recovery systems will likely be able to make use of insights and contributions of such a theory. It should be noted that despite the promise of control-theoretic techniques in analysis of system dynamics, they fall short of analysis of networked sensing systems. This is because computing systems offer new nonlinearities and different functionalities not adequately modeled by linear difference equations. Hence, extensions are needed to non-linear control to address the specific nonlinear and functional behaviors common to networked sensing systems in order to reason about their closed loop behavior. Such techniques must further be scaled to predict emergent behavior of large highly interconnected, interacting systems, as opposed to analyzing performance of isolated feedback loops.

1.6 Summary

This chapter described some practical considerations in the design of large networked sensing systems that arise in different interaction spaces between system components. Functional, data, temporal, and dynamic interaction spaces were explored. It was shown that new challenges arise in handling problems that occur by virtue of scale. Problems and interactions addressed in this chapter do not typically manifest themselves in smaller deployments. Tools and techniques are needed for sensor network designers to address the above composition and scaling challenges.

References

1. T. F. Abdelzaher, Y. Anokwa, P. Boda, J. Burke, D. Estrin, L. J. Guibas, A. Kansal, S. Madden, and J. Reich. Mobiscopes for human spaces. *IEEE Pervasive Computing*, 6(2):20–29, 2007.
2. D. Agrawal and C. C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *Proceedings of the 20th ACM SIGMOD Symposium on Principles of Database Systems*, pages 247–255, Santa Barbara, CA, 2001.
3. R. Agrawal and R. Srikant. Privacy preserving data mining. In *Proceedings of ACM Conference on Management of Data*, pages 439–450, Dallas, TX, May 2000.
4. J. Burke et al. Participatory sensing. Workshop on World-Sensor-Web, co-located with ACM SenSys, 2006.
5. Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. LiteOS, a UNIX-like operating system and programming platform for wireless sensor networks. In *IPSN/SPOTS*, St. Louis, MO, April 2008.
6. K. Chintalapudi and L. Venkatraman. On the design of mac protocols for low-latency hard real-time discrete control applications over 802.15.4 hardware. In *IPSN/SPOTS*, St. Louis, Missouri, 2008.
7. R. Cruz. A calculus for network delay, part i: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.
8. R. Cruz. A calculus for network delay, part ii: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991.
9. S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G.-S. Ahn, and A. T. Campbell. The bikenet mobile sensing system for cyclist experience mapping. In *SenSys '07: Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, pages 87–101, ACM, New York, NY, USA, 2007.
10. A. Evfimievski, J. Gehrke, and R. Srikant. Limiting privacy breaches in privacy preserving data mining. In *Proceedings of the SIGMOD/PODS Conference*, pages 211–222, San Diego, CA, 2003.
11. R. K. Ganti, N. Pham, Y.-E. Tsai, and T. F. Abdelzaher. Poolview: Stream privacy for grassroots participatory sensing. In *Proceedings of SenSys '08*, pages 281–294, Raleigh, NC, 2008.
12. P. Gupta and P. Kumar. The capacity of wireless networks. *IEEE Transactions on Information Theory*, 46(2):388–404, 2000.
13. T. He, B. M. Blum, Q. Cao, J. A. Stankovic, S. H. Son, and T. F. Abdelzaher. Robust and timely communication over highly dynamic sensor networks. *Journal of Real-time Systems*, 37(3):261–289, December 2007.
14. J. Heo, D. Henriksson, X. Liu, and T. Abdelzaher. Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study. In *Real-Time Systems Symposium*, Tuscon, AZ, December 2007.
15. J.-H. Huang, S. Amjad, and S. Mishra. Cenwits: a sensor-based loosely coupled search and rescue system using witnesses. In *Proceedings of SenSys*, pages 180–191, San Diego, CA, 2005.

16. Z. Huang, W. Du, and B. Chen. Deriving private information from randomized data. In *Proceedings of the 2005 ACM SIGMOD Conference*, pages 37–48, Baltimore, MD, June 2005.
17. B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. K. Miu, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In *4th ACM SenSys*, Boulder, CO, November 2006.
18. B. Hull et al. Cartel: a distributed mobile sensor computing system. In *Proceedings of SenSys*, pages 125–138, 2006.
19. C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, 2003.
20. P. Jayachandran and T. Abdelzaher. A delay composition theorem for real-time pipelines. In *ECRTS*, pages 29–38, Pisa, Italy, July 2007.
21. P. Jayachandran and T. Abdelzaher. Delay composition algebra: A reduction-based schedulability algebra for distributed real-time systems. In *RTSS*, pages 259–269, Barcelona, Spain, December 2008.
22. P. Jayachandran and T. Abdelzaher. Transforming acyclic distributed systems into equivalent uniprocessors under preemptive and non-preemptive scheduling. In *ECRTS*, Prague, Czech Republic, July 2008.
23. H. Kargutpa, S. Datta, Q. Wang, and K. Sivakumar. On the privacy preserving properties of random data perturbation techniques. In *Proceedings of the IEEE International Conference on Data Mining*, pages 99–106, Melbourne, Florida, 2003.
24. M. Khan, T. Abdelzaher, and K. Gupta. Towards diagnostic simulation in sensor networks. In *DCoSS*, Santorini, Greece, June 2008.
25. M. Khan, T. Abdelzaher, and L. Luo. SNTS: Sensor network troubleshooting suite. In *DCoSS*, Santa Fe, NM, June 2007.
26. M. Khan, H. K. Le, H. Ahmadi, T. Abdelzaher, and J. Han. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In *ACM Sensys*, Raleigh, NC, November 2008.
27. H. K. Le, D. Henriksson, and T. F. Abdelzaher. A practical multi-channel media access control protocol for wireless sensor networks. In *IPSN*, pages 70–81, St. Louis, Missouri, 2008.
28. H. Li, P. Shenoy, and K. Ramamritham. Scheduling messages with deadlines in multi-hop real-time sensor networks. In *11th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 415–425, San Francisco, CA, March 2005.
29. Microsoft Health Vault. http://www.healthvault.com/. Accessed on November 30, 2010.
30. T. Moscibroda. The worst-case capacity of wireless sensor networks. In *IPSN*, pages 1–10, Cambridge, MA, 2007.
31. J. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *IEEE Real-Time Systems Symposium*, pages 26–37, Madrid, Spain, December 1998.
32. N. Pham, R. Ganti, M. Y. Uddin, S. Nath, and T. Abdelzaher. Privacy-preserving reconstruction of multidimensional data maps in vehicular participatory sensing. In *European Conference on Wireless Sensor Networks (EWSN)*, Coimbra, Portugal, February 2010.
33. J. Song, S. Han, A. Mok, D. Chen, M. Lucas, and M. Nixon. Wirelesshart: Applying wireless technology in real-time industrial process control. In *11th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 377–386, April 2008.
34. A. N. Tikhonov and V. Y. Arsenin. *Solution of Ill Posed Problems*. V. H. Winstons and Sons, 1977.
35. K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Elsevier Microprocessing and Microprogramming*, 40(2–3):117–134, 1994.
36. Z. Yang, S. Zhong, and R. N. Wright. Privacy-preserving classification of customer data without loss of accuracy. In *Proceedings of SIAM International Conference on Data Mining*, pages 92–102, Newport Beach, CA, 2005.
37. N. Zhang, S. Wang, and W. Zhao. A new scheme on privacy-preserving data classification. In *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 374–383, ACM, New York, NY, USA, 2005.