Proceeings of the IEEE International Conference on Software Science, Technology and Engineering (SWSTE).
© 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained
for all other uses, in any current or future media, including reprinting/republishing this material
for advertising or promotional purposes, creating new collective works, for resale or redistribution
to servers or lists, or reuse of any copyrighted component of this work in other works.

Lessons and Experiences from the Design, Implementation, and Deployment of a Wildlife Tracking System

Sivan Toledo[†], Oren Kishon[†], Yotam Orchan[⋆], Adi Shohat[⋆], and Ran Nathan[⋆]
[†]Tel-Aviv University and [⋆]The Hebrew University of Jerusalem

***Abstract*—We describe software-engineering lessons we learned by building, deploying, and operating a large-scale distributed wildlife tracking system. The design started four years ago; the system has been operational for the past two years, but kept evolving during this time. The paper describes the structure of the system and then a series of interesting and well-documented lessons we learned. Most of the lessons surprised us, in spite of some of us being fairly experienced; some are not so surprising, but we felt that they are interesting enough to document here. Some of the lessons are particularly interesting because they are specific to computer systems built by computer scientists for collecting or processing experimental science data. These issues mostly revolve around the difficulty of building and maintaining complex systems in small teams in which junior members often leave well before the project is over.**

I. Introduction

The design and implementation of specialized computer system for experimental science offers computer scientists a unique opportunity but also unique challenges. Many scientific disciplines are increasingly dependent on sophisticated sensors, massive amounts of data, and sophisticated algorithms. These trends create opportunities for computer scientists to contribute to other branches of science. But contributing in a meaningful and substantial way requires overcoming significant challenges. Reliability and usability constitute one such challenge. For such a system to actually generate scientific results, it must be reliable and usable. Technology-demonstration prototypes often fall short of these criteria and fail to produce useful data for science. On the other hand, the total engineering effort that can be devoted to the system in an academic environment is usually quite limited. The evolving collaboration relationship between the experimental scientists and the computer scientists pose additional challenges.

This paper documents the main challenges that we faced and the main lessons that we learned in the course of a 4-year project that aims to develop a novel wildlife tracking system. The system, called ATLAS, has been deployed in the field gradually during the second year of the project and has been operational during the past two years. Development and additional deployment efforts also continued during the past two years. Our primary deployment currently consists of 3 beacon transmitters and 9 basestations (software-defined radio receivers) in remote locations in the Hula Valley in northern Israel, a server and database in Tel-Aviv University, and secondary servers and databases in the Hebrew University and in an Amazon cluster in Ireland. We have also successfully demonstrated rapid mobile deployment of a completely self-contained second system during a 2.5-day experiment in a rural area in another part of Israel. Recent analysis of the accuracy of ATLAS localizations shows that in the center of its coverage area, localization errors are of the same order of magnitude as the errors in GPS localizations [31]. More specifically, localization errors have typical standard deviation of about 5m and mean errors of about 5–15m.

The current implementation of ATLAS consists of more than 53k lines of code, about 38k of which in Java and the rest mostly in C.

The rest of the paper is organized as follows. Section II provides an overview of our system. We survey related work in Section III. The lessons we learned are presented in Section VI, and our high-level conclusions in Section VII.

II. An Overview of the System

Our system, called ATLAS, is a *reverse-GPS* system that localizes miniature transmitters attached to wild animals. In reverse-GPS systems, multiple receivers at known locations detect transmissions from transmitters in unknown locations. The receivers estimate the time of arrival (TOA) of each transmission and send these detection reports to a server. The server treats each detection report as a constraint of the form propagation-time = distance$/c$ where the propagation time is the difference between the (unknown) time of transmission and the time of arrival, the distance is between the unknown location of the transmitter and the known location of the receiver, and c is the propagation speed. The server estimates the location of the transmitter (and the time of transmission, which is useless but appears in the constraints) by minimizing the errors in a set of constraints associated with one transmission and several receivers. ATLAS uses radio signals that travel at the speed of light; over short distances, reverse-GPS systems can also use acoustic signals. This framework is called reverse-GPS because the role of transmitters and receivers in it are reversed from their roles in GPS, in which signals from transmitters in known locations (in space) are received and

This research was supported by the Minerva Center for Movement Ecology and by grants 965/15 and 863/15 from the Israel Science Foundation (funded by the Israel Academy of Sciences and Humanities).

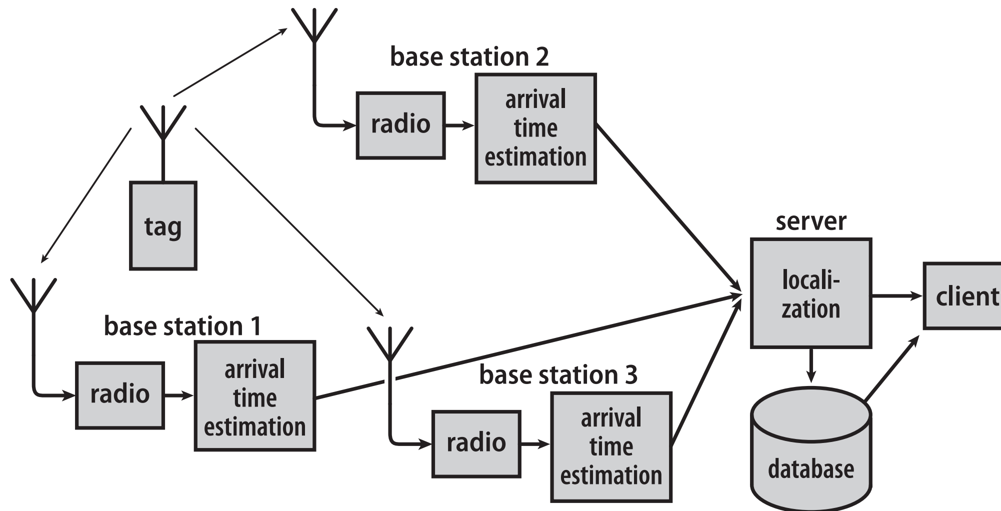

Figure 1. The overall structure of an ATLAS localization system. Basestations detect radio transmissions from tags and estimate their arrival times. A server estimates the location of tags from timed detection reports. Communication between basestations, the server, and clients use TCP/IP connections.

processed by receivers at unknown locations. The roles of transmitters and receivers are reversed in the two frameworks, but the physics is the same and the mathematical location-estimation frameworks are very similar. Figure 1 shows the structure of ATLAS.

ATLAS's transmitters, or *tags*, consist of a microcontroller and a low-power data transceiver [28]. They are powered by small batteries [26] and weigh 0.9g and up. Each tag transmits a unique long pseudo-random packet periodically, normally every second, 2s, 4s, or 8s. We have so far produced over 500 tags and deployed over 300 of them on wild animals, mostly birds and bats (and 50 more in the mobile-system experiment). We have collected over 39M localizations, including 51 individuals with $> 100,000$ localizations and 6 with $> 1,000,000$ (all Barn Owls carrying tags that weigh 10-15g). Altogether, we tagged individuals of more than 20 different species.

Transmission from tags are received and processed by *basestations* that consist of an antenna, low-noise amplifiers and filters [27], a sampling receiver, and a computer. Software running on the basestation's computer processes the RF samples, detects transmissions from tags, estimates their arrival time (to within a few nanoseconds to tens of nanoseconds), and sends the detection reports to the server. We use both commercial low-noise amplifiers and custom amplifier-filter units. The radio receivers are commercial units (USRP N200 by Ettus Research) and are fitted with a accurate and stable GPS-disciplined master oscillators; the accuracy and stability of the oscillators are critical to the performance of the system because we time the arrival of signals using the sample clock of the receivers. We synchronize the clocks at the basestations to a single time frame using transmissions from beacons (tags at fixed known locations). The detection of signals from tags and the arrival time estimation are done by correlating the received signal with a replica of the long pseudo-random packet that each tag transmits. The arrival-time estimate is obtained by interpolating the correlation values near their maximum; this yields an accurate sub-sample estimate. The cross correlation computation is expensive and accounts for most of the processing load in basestations. Due to this load, we mostly use relatively powerful computers in basestations (desktops with quad-core Intel i7 processors); in the mobile deployment, we successfully deployed weaker computers (laptops) who were still able to cope with about 50 tags. To reduce the computational load, the periodicity of tag transmissions is very accurate. Once a tag is detected, the system knows when it will transmit next (to within less than 100us). Basestations perform correlation computations only on windows of RF samples that contain subsequent transmissions along with margins of about 2ms. A significant portion of the code that a basestation runs is essentially a scheduler that schedules these digital-signal processing (DSP) computations. Most of the rest of the code performs the signal processing itself. The scheduler and configuration section of the code are written in Java; the DSP and the interface to the sampling receiver are written in C for performance.

Detection reports are sent by basestations to a *primary server*. These connections, like all other connections in an ATLAS system, use secure TCP connections. Our current deployments use cellular modems to provide connectivity to remote basestations. The connections between ATLAS components are completely reliable (messages are retransmitted periodically until acknowledged by the receiver) and they are delay tolerant (messages are eventually delivered to the destination even if the two sides crash and/or get disconnected).

The primary server groups detection reports of the same tag along with detection reports of a beacon from about the same time. These reports are used to form constraints in a weighted least-squares minimization problem whose solution is the estimated location of the tag. The detection reports and the localization solutions are written to an SQL database. The database also contains auxiliary tables that we describe later. The system has been tested with both MySQL and SQLite (a

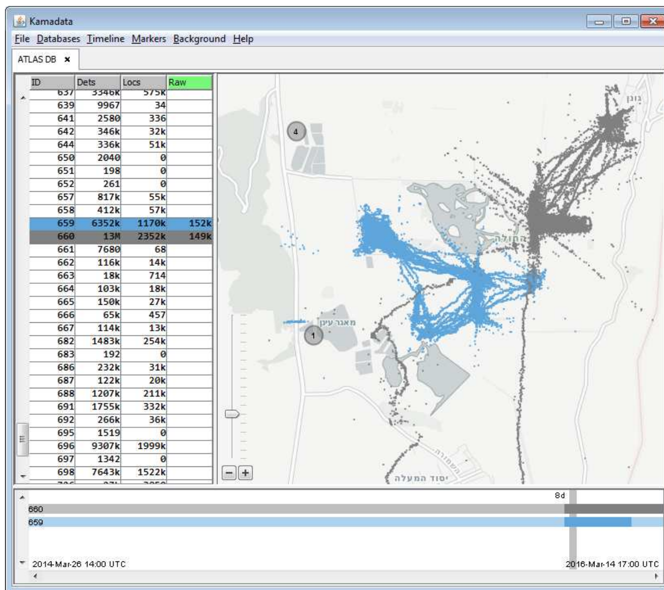

Figure 2. A graphical client showing raw localizations of two Barn Owls over four days.

serverless single-file database that requires less administration and configuration than, say, MySQL). The server can also send detection reports and localizations to clients. The server is implemented in Java. The non-linear minimization algorithms that solve for locations use building blocks from the Apache Commons Math library [9].

Real-time or *transient* clients receive all the detections and localizations that the server processes while the client is connected. *Persistent* clients are ones that the server knows about. Their connection to the server is reliable and delay tolerant, which means that after a down period or a disconnected period, the client receives all the data that it missed while down or disconnected. We currently support two different clients. One client is a graphical desktop program, shown in Figure 2, that can display both real-time data and localizations stored in the database; it is a transient client. *Secondary servers* are clients that run essentially the same code as the primary server; the only difference is that they do not receive detection reports directly from basestations; they receive all the data (detections and localizations) from another server.

Servers, both primary and secondary, can be configured in a variety of ways. They can utilize a unified database or no database, or a separate SQLite for every day; these daily files constitute an application-level backup mechanism. A server can be configured to compute localizations or not, to support a set of persistent clients, to compute summary tables (statistics) and presentation tables (containing the most recent data for display on a web site), and so on.

Delay tolerance in the servers is not limited to reliable transport of data. When a detection report arrives hours or days after the actual event, the server recomputes the localization that was derived from the same transmission (in some cases, it computes the localization for the first time, if there were not enough constraints without the late detection report). The re-computation is not done upon the reception of every late report, which would be expensive in terms of database queries. Instead, the late report marks a time-span containing the arrival time (typically one hour) as *invalid* in an auxiliary table. A clean-up thread checks for these invalidation records and recomputes a batch of localizations if one is found.

ATLAS components that must run continuously (basestations and servers) are executed by a script that runs them in an infinite loop, with a delay of 10s between invocations. The delay aims to limit overhead and to allow missing system resources that might have caused the component to crash (e.g., connections to the radio or the the Internet) to be restored before ATLAS is restarted. These scripts are typically invoked automatically from `/etc/rc.local` after the computer boots.

Two additional components are used on an ad-hoc basis, not on a continuous basis. The *query* program queries an ATLAS database. It can recompute localizations from detections, and it stores the output of the query or of the localization computation in a variety of ways: in a text table for analysis in Matlab, in a standalone SQLite ATLAS database file, or back in a unified ATLAS database. Running the localization algorithm in this way allows the use of estimation formulations that are more expensive and potentially more accurate than the fast formulation used by default by ATLAS servers. Localizations computed off-line by this program also include estimates of uncertainty (covariance matrices, residuals, and gradient norms) that the servers do not compute.

The *homing-in* program uses the basestation code base to receive transmissions from a single specified tag and to present the signal strength of the received signals both visually and as an audible tone. This program is used to find the direction to a tag (with a directional antenna that can be rotated) in order to retrieve it even in locations with no ATLAS coverage. This retrieval technique, called homing in, has several applications in wildlife research and management, such as retrieving expensive tags or data-logging tags. ATLAS tags transmit wideband signals that existing homing-in receivers cannot process, thereby necessitating this program. We also use it to test tags prior to deployment.

ATLAS is highly configurable and is designed to support many instances at different sites. The configuration for a particular site is described by a set of configuration files that are stored in one directory on a web server. Some of the files contain Java properties (text files with key-value pairs that the system can query) that configure various ATLAS components and others contain plain-text representation of Java objects that represent the particular site: the tags used in the site, the tags that basestations need to track, and the location of beacons and basestations. The web server in which we store these files is actually a Subversion server; users update the configuration files of their site using a Subversion client. This scheme allows simple HTTP access to the files from ATLAS components, but at the same time provides version management to users. To allow ATLAS components, especially basestations, to boot

even when disconnected from the Internet, ATLAS caches these files in the current working directory. When a component starts, it checks if the server has a newer version of one of the files and if so replaces the old version by the most recent one. If the server is not accessible but the files are present, the component starts anyway. Components also check the server for a new version of these files every few minutes. If the local file that was used at startup time becomes obsolete, the code exits; when the script restarts it 10s later, it retrieves the new version.

III. Related Work

The field of *movement ecology* is a branch of life sciences that studies ecological processes primarily through analysis of the movement of individuals and ensembles of animals and plants [25]. This field has been undergoing rapid growth in recent years thanks to new technologies to track animals and thanks to computer systems that can process large amounts of tracking data [4], [6], [16].

This growth has been driving research on the engineering and deployment of both the tracking systems and the data processing systems. We list several key examples. Kays et al. [15] document experiences from several years of operation of a regional-scale angle-of-arrival (AOA) wildlife tracking system in a nature reserve in Panama. Dyo et al. [10] document software and hardware evolution during a year-long project that tracked Badgers using RFID tags. They also describe a variety of deployment challenges and solutions. Juang et al. [14] describe software, hardware, and deployment issues of ZebraNet, a tracking system for large social mammals.

More generally, building and deploying distributed sensing systems is hard; a fairly large number of reserach papers have been devoted to describing system building and deployment experiences and the lessons that can be learned from them (see, e.g., [2], [13], [19], [24]).

Interest and research on management of large-scale animal-movement databases is also growing [18], [29].

ATLAS as a system is related to a number of earlier systems described in the literature. The closest one is a pioneering reverse-GPS wildlife tracking system built by MacCurdy et al. [21]. In spite of this system's novelty, it has been used very little to produce science results, perhaps as a result of engineering that aimed primarily to prove that the concept is viable as opposed to long-term production use by scientists. The angle-of-arrival system that Kays et al. [15] is based on a different physical principle than ATLAS, but it is similar to ATLAS in the senses that it was a distributed system, it was engineered as a production system (also by a small team), and was in operation in a remote location for several years.

IV. Team-Related Challenges in the Development of Systems that Support Experimental Science

This section discusses personnel-related challenges in the development of complex computer systems that support experimental science. Such systems are developed by three types of teams. Some systems are developed by an experimental-science group (e.g, [21], developed primarily by a staff engineer in an Ornithology research group). Others, like ATLAS, are developed by computer scientists, often in collaboration with prospective users; for other systems in this category, see [10], [14], [22], [23]. The rest are developed by commercial companies; LimeLight[1], a system for tracking animals in an arena and quantifying their movement, examplifies systems in this category.

ATLAS was designed and built by a team consisting of both computer scientists and life scientists in academic institutions. Some of the challenges and lessons that we discuss are also relevant to other types of teams, but some are specific to this collaborative setting.

A. *The Challenges of Small Teams*

Most of the science-support systems built in academic environments are designed and implemented by small teams. This appears to be true for most of the systems mentioned above, both those that were built by the experimental-science group and those built in collaboration with computer scientists; it is probably also true for many commercial systems, especially specialized ones that serve only a small community of scientists.

The small size of a team is not an impediment when the system is not particularly challenging, or when the design and implementation challenges are limited to one or two areas in which the small team has expertise (see, e.g., [1]). However, when the technical challenges are frequent and diverse, the small team size is constraining in at least two senses.

First, small teams typically do not possess expertise in many different areas. Learning to gain expertise is possible, of course, and is often necessary to carry out the project. But this takes time and effort, so the learning processes must be limited to areas that deserve the effort and to the level of expertise that is actually required. These decisions are difficult to make, because by definition they are made when the team lacks expertise in a problem area. Sections VI-A and VI-B document decisions of this type that we have made, both good and bad.

Second, a small team typically lacks sufficient engineering resources. This can easily lead the team to cut corners, especially on software engineering practices that appear to bring little immediate benefit, such as testing, documentation, and rigorous definition of interfaces. When this is coupled with motivation and incentives that are not aligned with system reliability, the severity of the problem increases; we discuss this topic in Section IV-C below.

B. *Two Kinds of Funding Shortages*

Research projects to develop novel complex systems like ATLAS are challenging to carry out in academic institutions due to funding that is limited in two different ways, both significant. First, the amount of funding is limited, leading to

[1] http://actimetrics.com/products/limelight/

small team sizes and often also to inability to hire experienced engineers. Second, typical research grants provide funding for only a few years, roughly the duration of a PhD project, making initiation of longer-term projects risky.

C. Motivation Discrepancies in Collaborative Projects

By definition, the main motivation for a science-support system is the production of scientific results. In a project carried out by the experimental-science group, this is often the only motivation. But in a collaborative project, the computer science (or electrical engineering) group is usually driven by a strong secondary motivation: to advance its own field and to publish in its venues. Without this motivation, the computer science group is unlikely to contribute in a significant way to the design and implementation of the system (it may contribute in a minor way by giving advice).

The motivating factors of the two groups converge in areas that represent a technological challenge. For example, if it is difficult to product accurate results (this has been one challenge in ATLAS), then both groups are motivated to address the challenge: the experimental-science group needs accurate measurements, and the computer-science group wants to solve the estimation or system challenge. The same might be true for capacity challenges: the experimental scientists may need large quantities of data, and the computer scientists may welcome the performance challenge.

But the motivating factors diverge in other areas, especially ones related to robust software engineering. The experimental scientists usually need a system that is robust, reliable, usable, and maintainable. These traits require an engineering effort that the computer scientists are not rewarded for.

In other words, technological innovation rewards both sides of the collaboration, but computer scientists are best served by a proof-of-concept system, whereas the experimental scientists are best served by a well-engineered production system. In particular, building production-quality systems entails two costs that are hard to justify in computer science research projects. One consists of engineering activities (e.g., extensive testing and documentation) that computer scientists, including graduate students, postdocs, and faculty members usually receive no credit for; they are measured by innovation and analysis, not by quality metrics of the software that they write. The other is maintenance; production systems must be maintained, and by the time non-trivial maintenance is required, the students and postdocs who designed and implemented the system are usually long gone. We note that in some cases graduate students write useful code and continue to maintain it for many years, but this is the exception rather than the rule. FFTW, an FFT library, is a good example of such software [12].

D. Interdisciplinary Cultural Differences

There are core structures and values that are shared by many academic disciplines, but there are also many differences between disciplines. These differences create tensions in collaborative projects. We encountered many surprises along the way. Computer scientists often publish relatively quickly, often within a year or so of the beginning of a project, whereas ecologists often take longer to obtain sufficiently large sample size and/or enough repetitions across seasons (for example), and to cope with unexpected problems typical of any fieldwork. When computer scientists promise to deliver software in the context of a research project, they usually mean that they will deliver a flaky prototype; when life scientists imagine software products, they imagine robust industrial-strength software. We could go on and on.

E. Mitigation and Guidance

We now describe how we addressed the challenges listed in Sections IV-A to IV-D, and we propose additional coping mechanisms.

Small Teams: Most of ATLAS's code was written by the first two authors, a faculty member and a graduate student. Some of the algorithmic code was based on prototype codes written by three additional students, two graduate students and an undergraduate. The majority of the code written by the second author and all the code written by the other students was subsequently re-written by the first author, a faculty member, to improve code quality metrics and to reduce concerns over long-term maintainability. Our assessment is that this solution addressed the code-quality and maintenance issues, as well as some of the motivation discrepancies (more on that below). The costs to the faculty member in terms of reduced productivity (e.g., publication count) are hard to assess, but this is certainly a valid concern. In other words, this is probably not a solution that is universally applicable.

We also tried to address this issue, at least partially, by delegating responsibility to software modules that seemed less interesting (primarily the web interface to the system) to the experimental-science group, which employed programmers to design and implement these modules. Our assessment is that this did not work well. The programmers employed the by science group were all junior (undergraduate computer-science students or people who just received a computer-science degree), they worked in an environment with no professional mentoring, and most of them persisted in this job a relatively short period of time. We do not recommend this approach.

Another way to address these challenges is to employ staff engineers to design and implement the production-quality version of the system. We did this, but fairly late in the project, mostly due to the time lag it took to raise sufficient funding (but also because we did not fully understand the utility of this early on).

Funding: We started the project only after securing stable long-term funding. In 2012, the last author received long-term funding (6+6 years) from the Minerva Foundation and the Hebrew University to launch a research center in movement ecology. This was a key turning point because it dramatically reduced the risk of running out of funding before the new system matured enough to deliver useful science results. Consequently, we conceived and launched the ATLAS project as a joint initiative of the two PIs (the first and last authors) under the auspices of the new Minerva Center for Movement

Ecology (MCME). Research centers like the MCME, which are rather rare in academic institutes, provide the means to advance relatively long-term projects. In our case, the Minerva Center funds were sufficient to recruit a technical manager responsible for all implementation issues and fieldwork (the third author), to hire several students and research assistants (including the second and fourth authors), to purchase the required equipment and to cover associated costs (e.g. communication and maintenance). Our preliminary work in the first two years enabled us to secure additional funds in the form of a joint research grant from the Israeli Science Foundation. This grant supports two students, several part-time assistants and two post-doctoral fellows in both groups. Funding remains a limiting factor, especially with regards to recruiting full-time experienced engineers for extended periods.

For non-trivial collaborative systems, securing long-term funding is essential. We note that the last author envisioned a collaborative project similar to ATLAS more than a decade ago, but those plans did not materialize due to lack of sufficient funding.

Motivation and Incentives: We made the motivation of each group clear and explicit early on and agreed how to share credit in a way that provides sufficient incentives to all parties. We feel that this is absolutely essential, even if it does not completely eliminates tensions.

Summary: Our main recommendation is that teams that initiate design of new systems in support of experimental science be aware of these challenges and plan their projects accordingly. The fact that a certain party (individual or team, in a computer-science group or in an experimental science group) can design and implement a system does not imply that the system will engineered well enough to produce useful science results. Similarly, an agreement to collaborate intensively on an interdisciplinary long-term project does not completely eliminate surprises and tensions down the road; they are bound to happen so participants should expect them.

V. Doing Research while Developing: Prototyping Tradeoffs

The software engineering practices that are used in collaborative projects that aim to develop systems for experimental scientists need to support both technology-focused research and efficient construction of robust software. One practice that is particularly important to both tasks is the use of software prototypes, implementations that are relatively easy to build. They are used to demonstrate feasibility of software components and to explore the design space. The utility of prototypes has been recognized decades ago [5]. The tradeoffs associated with with prototypes in general and with different kinds of prototypes (e.g., throw-away vs. evolutionary) continue to be discussed in the software engineering literature [7], [17], [20], but in general they are well understood.

Science-support systems built in collaboration with computer scientists tend to be high-risk; if they are straightforward, there is no motivation for computer scientists to collaborate. The feasibility-demonstration aspect of prototypes is particularly important in challenging high-risk systems, so they are useful in systems like ATLAS. We indeed used two major software prototypes (as well as many hardware prototypes [28], [27]); we discuss them later in this section.

The time and effort required to build both a prototype and a production system are a potential disadvantage of prototypes. The best tools for building prototypes are sometimes inappropriate for production use. This may be due to low performance, to lack of reliability-enhancing mechanisms like strong typing, or to high cost. When such tools are used, the prototypes tend to be of the *throw-away* type. When the prototype is an incomplete implementation built with tools that are also appropriate for production, the prototype can evolve into a production implementation. So-called *evolutionary prototypes* can lead to lower overall cost than throw-away ones because less code is thrown away. Minimizing the total effort is particularly important when a small team builds a challenging system.

Prototypes are often good research platforms. Development tools that enable rapid prototyping also enable rapid instrumentation, modification, and sometimes visualization. This implies that the prototype may better support technology-focused research investigations than the production implementations that replace them.

We built prototypes of both the basestation and the server programs of ATLAS. Both prototypes used Matlab implementations of numerical algorithms and Java classes for communication, task queues, and non-numerical data structures (e.g., for batching detection reports into sets of detections of the same transmission). The prototype of the basestation software also included a C program that communicated with the Matlab/Java program through a TCP connection. The role of the C program was to configure the radio receiver and to collect and buffers samples from it. The Matlab/Java basestation program performed digital signal processing (DSP) to detect transmissions from tags and to estimate their arrival times. The server prototype was also split into two programs (processes), a pure Java program that communicated with basestations and stored data in a database and a Matlab/Java program that computed localizations.

We replaced the DSP code that was implemented in Matlab with a C implementation just prior to the deployment of the first basestation. The main rationale was to eliminate the need to manage Matlab licenses in remote machines. We later also replaced the separate radio-interface C program with a version of the code that is callable from Java and runs in the same process. This was done in order to improve stability and to enable operators to diagnose and fix problems more easily. Code stability and licensing concerns also led us to replace the Matlab implementation of the localization algorithms with a Java implementation. Our Java code does not implement the localization algorithms directly; it relies heavily on numerical algorithms implemented by the Apache Common Math library.

A. Prototypes: Lessons and Recommendations

Our assessment is that using Matlab or a similar tool for the initial implementations was the right thing to do. Matlab offered programming environment with a large and well documented library of sophisticated algorithms and powerful visualization tools. These properties allowed us to to build working software relatively quickly. We also feel that the decision to replace the Matlab implementations with C and Java ones led to better engineered code that is more stable. The Java implementation of the localization code is also well structured so it should be easier to maintain than the initial Matlab code. The performance of the C signal-processing code is critical to the performance and capacity of ATLAS basestations. As a result, the code is low level and complex. It is likely to be much more difficult to maintain than a comparable Matlab code.

We did not explicitly decide upfront that the Matlab implementations would only serve as prototypes. This was probably a mistake, because it caused us to spend time on careful engineering of Matlab codes that were eventually discarded.

An important and unexpected consequence of the transition from a Matlab prototype to a C or Java production code was that visualization and algorithmic investigation became more difficult. We did maintain both implementations for a while, in order to preserve the visualization and experimentation capabilities. But as the software continued to evolve, we realized that maintaining two working versions of a components was too costly and we ceased to maintain the Matlab implementations. It is hard to determine whether we actually lost some research opportunities when we replaced the Matlab implementations with Java and C. However, making visualization and experimentation more difficult is clearly a significant disadvantage in a system that is a computer-science research project as well as a science-support system.

Our recommendations to teams developing similar systems are (1) to use rapid prototyping environments for at least the initial implementation of challenging components; (2) to try to decide early on which implementations are throw-away prototypes and which will evolve into production code; (3) to carefully weigh the costs and benefits of throwing away and replacing prototypes. The costs consist not only of the additional implementation effort, but also of more difficult maintenance of complex high-performance implementations (especially in low-level languages like C) and of the increased difficulty of algorithmic experimentation and visualization. The benefits of license-freedom and increased stability and performance may or may not justify replacing the prototype. Careful initial planning may also point to programming environments that are suitable for both prototyping and production, eliminating the need to throw away prototypes. Having said that, it is also true that the benefits of prototyping include the freedom to defer decisions about implementation languages and environments, a freedom that must be given up if the prototype is to evolve into the final product.

VI. Generic Software Engineering Challenges and Lessons

Not surprisingly, we have also encountered during the project challenges that are not specific to systems designed to support experimental science. The first two issues that we discuss in Sections VI-A and VI-B are particularly important to small teams; the other issues that we discuss later in the section are generic.

A. Do-it-Yourself or Ready-Made?

One tradeoff that comes up again and again is whether to use existing software modules (libraries or entire subsystems) or to develop the required functionality from scratch. The benefit of using an existing library/subsystem is clear: the functionality of the module becomes available without any design or implementation effort. But there are also costs to consider. The costs that worried us most are (1) future maintenance tasks due to the long-term dependence on the modules that are used, (2) the effort to choose a module if more than one is available, and (3) the effort required to learn how to use the module effectively and correctly. In the case of commercial modules, the monetary cost is obviously also an issue. The literature usually refers to this issue as the *build-or-buy* decision and to the existing subsystem as a COTS (commodity off the shelf) product.

The maintenance cost induced by dependence on a COTS module is hard to assess, mostly because this cost is paid long after the decision to use the module. We were worried about these costs and we did make a conscious effort to avoid dependence when the payoff seemed small. More specifically, we were concerned that future upgrades of modules that we depend on will modify their behavior in a way that requires changes in our code. We were also concerned about the opposite problem, that lack of maintenance of a module will cause problems.

Obviously, when an existing module offers sophisticated functionality that is actually needed, the right thing is to use it. The tradeoff requires consideration mostly when the functionality is relatively simple.

This issue came up several times during the development of ATLAS. One interesting case is the logging infrastructure. We realized that we need a logging module when we started running the basestation code continuously. We were aware that there exist logging libraries for Java, but at the time we felt that the functionality is so simple that it would be easier to develop it ourselves than to learn to use a logging library. We developed a class that performed logging to a file and used it for well over a year. When we realized that we need to constrain the size and number of log files, we switched to using the Apache Commons Logging library [8]. The switch required an effort to select this particular logging library, to learn how to configure it and to use it, and to modify all the call sites (we decided to call the new library from the calls sites directly in order to take advantage of the severity-level support rather than to use the old class as a wrapper). The original coding effort was

thus wasted; in hindsight, it would have been better to use an existing logging library immediately.

This issue came up several more times. The most interesting cases are listed below.

- We use Subversion, a source-code management system[2], to manage the configuration files of deployed ATLAS systems. The system retrieves up-to-date configuration files through secure HTTP connections (Subversion running under Apache offers this access method to the most up-to-date version of the file, in addition to access though a custom protocol). System administrators update configuration files using the Subversion interface, which offers sophisticated versioning and authorization capabilities. The main costs associated with the use of this subsystem are the need to install a subversion client on ATLAS administrators' machines and the need to deploy a Subversion server as part of ATLAS installations (other file repositories with an HTTP retrieval mechanism should also work).
- We use the Apache Common Math Java library [9] to solve the nonlinear optimization problems whose solutions define locations estimates, and also to perform linear algebra transformations and decompositions that are associated with the localization algorithms [31].
- On the other hand, we have implemented the signal processing that is used for detection of signals emitted by tags and for estimating their arrival time in custom code written in a combination of Java and C (mostly C). The C code does call a high-performance Fast Fourier Transform (FFT) library, FFTW [12], but the rest of the code is custom. We have investigated a few digital-signal-processing libraries (e.g., GNU Radio [3]) but did not find one that was appropriate. The custom code that we wrote is fairly simple algorithmically, but it uses complex memory management in order to achieve high performance. The decision to use an existing FFT library was simple: the performance of the arrival-time estimate code depends mostly on the performance of the FFT algorithm that it calls, so using an existing high-performance library is essentially the only reasonable option. The interface to FFTW is fairly complex so the decision to use it does carry a cost in terms of code clarity and simplicity.
- We also decided not to use an existing delay-tolerant reliable transport layer. We have not investigated this issue deeply, but from the superficial investigation that we did carry out no easy-to-use and easy-to-deploy candidate emerged. We implemented our own custom transport layer.

B. Hidden Complexity

We mentioned above the effort required to learn how to use a sophisticated module effectively and correctly. In one case, we introduced a serious bug into the system because we failed to understand the complexity of one simple-looking module. Finding the fault was difficult. Fixing it was easy.

The module in question is the Java serialization mechanism (`java.io.ObjectInputStream`, `ObjectOutputStream`, and related classes and interfaces). This mechanism is part of Java's standard library, but using it correctly is not trivial. The bug in our code caused long-running ATLAS programs (the basestation and server programs) to run out of memory. A memory-contents analysis revealed that most of the heap stored objects that were sent or received by object streams. Reading the documentation carefully, we realized that the streams maintain a reference to every object that passes through them, in case another reference to one of them reappears in a serialized object. This behavior is required in order to duplicate aliased structures on the receiving side. Once we understood what caused the problem, we fixed the bug by replacing the calls to the `read` and `write` methods by calls to `readUnshared` and `writeUnshared`, which do not maintain references to the objects (and do not guarantee correct reconstruction of aliased structures).

In Java and other languages with a vast standard library, it is prudent to expect that some sections of the standard library would require the learning curve that we normally associate with complex external modules and libraries.

C. Data Transport Reliability: Near-Perfection is Hard to Explain, Perfection is Unforgiving

The first version of the delay-tolerant transport layer that we designed and implemented seemed to satisfy all the requirements of the system. It was not 100% reliable but came close; it was allowed to rarely lose messages. This was justified by the argument that lost messages are primarily real-time arrival-time reports that can also be lost due to a variety of other reasons: RF noise and interference, software upgrades and restrarts, and so on.

In the almost-perfectly-reliable design, outbound messages are placed in a persistent file-backed queue. If the TCP connection is too slow to cope with the sending rate or if the remote side is unreachable, the in-memory queue spills into a file. Messages are never discarded because the queue is full. A specialized thread tries continuously to extract messages from the in-memory queue or from the file and to write them to the TCP connection. The only messages that are lost in this design are the ones that are written to the outgoing TCP connection but are not delivered because the connection disconnects. Assuming disconnections are rare, relatively few messages are lost. We did experience a massive loss of messages during a period of a few days in which the server crashed repeatedly due to an unrelated bug, causing TCP connections to be established and disconnected repeatedly. But except for this incident, the system did deliver most of the messages.

The main defect in the almost-perfect design was that it proved difficult to explain the level of data-transport reliability to users and potential users. As soon as they understand that ATLAS collects data from computers deployed at remote

[2]https://subversion.apache.org/

locations, they ask what happens to the data that is collected when a basestation is disconnected. The original design did not allow us to respond that all of this data always eventually arrives at the server. The explanation that we provided was complex and appeared to be confusing to many users.

To address this defect, we improved the design so as to make data collection and processing perfect, not just almost perfect. In the new design, messages (objects) to be sent are collected into *bundles* that are acknowledged as a unit by the receiver, to reduce the overhead of sending and processing acknowledgments. A bundle is finalized and transmitted when the oldest message in the bundle passes an age threshold to limit latency (30s by default), or when the bundle passes an object-count threshold. Bundles that need to be sent through a TCP connection are first stored in an SQLite database associated with the connection. Bundles are deleted when they are acknowledged by the receiving side and are retransmitted periodically as long as they are not acknowledged.

An ATLAS server only acknowledges an incoming bundle after the objects in the bundle has been stored in all the databases that the server maintains. This includes the main ATLAS database maintained by the server and also the SQLite databases associated with outgoing connections to clients or secondary servers.

The switch to the completely-reliable design caused unexpected failures. The failures were due to limited capacity in the reliability-ensuring mechanism, namely the SQLite database. If bundles are too small, the database may not be able to commit them at a sufficient rate. Interestingly, this potential bottleneck is in the system was introduced in order to ensure message reliability; in the almost-perfect design, this potential bottleneck did not exist. The failures were also hard to diagnose, because the limited commit rate slows down the module that sends messages, which slows down the modules that feed it, and they slow down the modules that feed them, and so on. Back pressure causes large parts of the system to slow down. It was hard to pinpoint what was slowing down the system. We eventually discovered and corrected the problem. But in general, completely-reliable systems operate without pressure-escape valves, which means that performance bottlenecks must cause failures, and the failures are not necessarily close to the bottleneck, making diagnosis difficult.

D. Object-Oriented Zeal

ATLAS is a distributed system that sends objects (messages) of several types between system components. In procedural programming environments, messages carry a type field (e.g., the tag in MPI messages [11]). The receiver of a message inspects the type field and processes the message accordingly. The first design of ATLAS's messaging mechanism was based on this idea. Objects were serialized and sent to the destination. The receiving thread deserialized the object and called a handler method. Conditional statements in the handler checked the type of the received object using the `instanceof` operator and processed the message. Figure 3 (left) presents this design.

It worked, but we decided to use a more elegant solution, shown on the right side of Figure 3, which avoids the `instanceof` operator and any other explicit message-type tag. All messages have a common denominator: they must be processed at the receiving side in some way. This was expressed using a `Message` interface with one method, `execute`, which is invoked by the receiver of a message and it is responsible for processing the message. In this design, the code in a class that implements `Message` is executed in two completely different run-time environments: the constructor is executed at the sender, whereas `execute` is executed at the receiver. The receiver runs the correct code using Java's dynamic dispatch mechanism, not by explicitly checking the type with `instanceof`. Our initial use of `instanceof` is indeed considered in the software-engineering literature to be a manifestation of poor object-oriented practices; and the dynamic-dispatch design we used later is the recommended solution [30].

This worked well as long as ATLAS had only two kinds of communicating components, basestations and a single server. Messages were constructed at basestations and their `execute` were executed on the server.

Over time, we developed additional ATLAS components: secondary servers, a command-line query interface, real-time visual clients, and the homing-in program. These components also receive messages, but a message received by one of these components needs to behave differently than it does at the main ATLAS server. For example, a detection message arriving at the main server is stored in a database and grouped with others to form a localization problem, but the same message arriving at a real-time visual client updates a table of recent detections. The original `execute` method did not express all of these behaviors; to express all of them in this method would again require conditionals to determine the execution environment.

We reverted to the original design. Every ATLAS component that receives messages has a handler method that tests the type of incoming messages and processes appropriately.

VII. Conclusions

Building a computer system that enables experimental scientists to collect new kinds of data or collect data in new ways can be exciting, rewarding, and productive for computer scientists. Such a project can both advance science and open up interesting opportunities in systems research.

We have learned a lot from building ATLAS, a novel distributed system for tracking wildlife. Some of the lessons are generic and can be learned from and applied to many other types of computer systems. Other lessons are specific to science-support systems that are developed by academic teams (or teams in other types research institutions). In such cases, the system is being used as a production system by the experimental scientists, but the development is done in a team that rarely or never delivers production systems. The discrepancy creates many challenges that stem from the small size of the development team, from the limited tenure of junior

```
public void handle(Message m) {
  if (m instanceof DetectionMessage) {
    DetectionMessage dm = (DetectionMessage) m;
    ... // process an incoming detection message
  }
  if (m instanceof (TagSummaryMessage) {
    ...
  }
  ...
}
```

```
public class DetectionMessage implements Message {
  public final int basestation;
  ...   // other fields
  public DetectionMessage(...) {...}
  @Override public void execute() {
    ... // process an incoming detection message
  }
  ...   // other methods
}
```

Figure 3. Two ways of handling incoming messages in an object-oriented program.

team members, from funding cycles that are too short, and from friction between developers and users. We have tried several techniques to mitigate the challenges; most worked, some did not.

References

[1] Vsevolod Afanasyev. A miniature daylight level and activity data recorder for tracking animals over long periods. *Memoirs of the National Institute for Polar Research Special*, 58:227–233, 2004.

[2] Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, and Martin Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 43–56, 2008.

[3] Eric Blossom. GNU Radio: Tools for exploring the radio frequency spectrum. *Linux Journal*, June 2004.

[4] E. S. Bridge, K. Thorup, M. S. Bowlin, P. B. Chilson, R. H. Diehl, R. W. Fléron, P. Hartl, K. Roland, J. F. Kelly, W. D. Robinson, and M. Wikelski. Technology on the move: Recent and forthcoming innovations for tracking migratory birds. *BioScience*, 61:689–698, 2011.

[5] Fred Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[6] F. Cagnacci, L. Boitani, R. A. Powell, and M. S. Boyce. Animal ecology meets GPS-based radiotelemetry: A perfect storm of opportunities and challenges. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 365:2157–2162, 2010.

[7] Alan M. Davis. Operational prototyping: a new development approach. *IEEE Software*, 9:70–78, 1992.

[8] Commons Logging Developers. Apache commons logging, release 1.2. Available from https://commons.apache.org/logging, 2014.

[9] Commons Math Developers. Apache commons math, release 3.5. Available from https://commons.apache.org/math, 2015.

[10] Vladimir Dyo, Stephen A. Ellwood, David W. Macdonald, Andrew Markham, Niki Trigoni, Ricklef Wohlers, Cecilia Mascolo, Bence Pásztor, Salvatore Scellato, and Kharsim Yousef. WILDSENSING: Design and deployment of a sustainable sensor network for wildlife monitoring. *ACM Transactions on Sensor Networks*, 8:29:1–29:33, 2012.

[11] Message-Passing Forum. MPI: A message-passing interface standard. Technical report, 1994.

[12] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[13] Timothy W. Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I. Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. The hitchhiker's guide to successful residential sensing deployments. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 232–245, 2011.

[14] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with ZebraNet. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 96–107. ACM, 2002.

[15] R. Kays, S. Tilak, M. Crofoot, T. Fountain, D. Obando, A. Ortega, F. Kuemmeth, J. Mandel, G. Swenson, T. Lambert, B. Hirsch, and M. Wikelski. Tracking animal location and activity with an automated radio telemetry system in a tropical rainforest. *The Computer Journal*, 54:1931–1948, 2011.

[16] Roland Kays, Margaret C. Crofoot, Walter Jetz, and Martin Wikelski. Terrestrial animal tracking as an eye on life and planet. *Sciene*, 348:aaa2478–1–aaa2478–9, 2015.

[17] Fabrice Kordon and Luqi. An introduction to rapid system prototyping. *IEEE Transactions on Software Engineering*, 28:817–821, 2002.

[18] B. Kranstauber, A. Cameron, R. Weinzerl, T. Fountain, S. Tilak, M. Wikelski, and R. Kays. The Movebank data model for animal tracking. *Environmental Modelling and Software*, 26:834–835, 2011.

[19] Lakshman Krishnamurthy, Robert Adler, Phil Buonadonna, Jasmeet Chhabra, Mick Flanigan, Nandakishore Kushalnagar, Lama Nachman, and Mark Yarvis. Design and deployment of industrial sensor networks: Experiences from a semiconductor plant and the north sea. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 64–75, 2005.

[20] Phillip A. Laplante and Colin J. Neill. The demise of the waterfall model is imminent. *Queue*, 1:10–15, 2004.

[21] R. MacCurdy, R. Gabrielson, E. Spaulding, A. Purgue, K. Cortopassi, and K. Fristrup. Automatic animal tracking using matched filters and time difference of arrival. *Journal of Communications*, 4(7):487–495, 2009.

[22] Andrew Markham, Niki Trigoni, Stephen A. Ellwood, and David W. Macdonald. Revealing the hidden lives of underground animals using magneto-inductive tracking. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 281–294, 2010.

[23] Andrew C. Markham and Andrew J. Wilkinson. EcoLocate: A heterogeneous wireless network system for wildlife tracking. In Tarek Sobh, Khaled Elleithy, Ausif Mahmood, and MohammadA. Karim, editors, *Novel Algorithms and Techniques In Telecommunications, Automation and Industrial Electronics*, pages 293–298. Springer, 2008.

[24] Robert S. Moore, Bernhard Firner, Chenren Xu, Richard Howard, Yanyong Zhang, and Richard Martin. Building a practical sensing system. In *IEEE iThings*, 2013.

[25] Ran Nathan, Wayne M. Getz, Eloy Revilla, Marcel Holyoak, Ronen Kadmon, David Saltz, and Peter E. Smouse. A movement ecology paradigm for unifying organismal movement research. *Proceedings of the National Academy of Sciences of the United States of America*, 105:19052–19059, 2008.

[26] Sivan Toledo. Evaluating batteries for advanced wildlife telemetry tags. *IET Wireless Sensor Systems*, 5:235–242, 2015.

[27] Sivan Toledo. A selective robust weak-signal UHF front end. *QEX*, pages 31–36, January/February 2015.

[28] Sivan Toledo, Oren Kishon, Yotam Orchan, Yoav Bartan, Nir Sapir, Yoni Vortman, and Ran Nathan. Lightweight low-cost wildlife tracking tags using integrated transceivers. In *Proceeings of the 6th Annual European Embedded Design in Education and Research Conference (EDERC)*, pages 287–291, 2014.

[29] Ferdinando Urbano and Francesca Cagnacci. *Spatial Database for GPS Wildlife Tracking Data: A Practical Guide to Creating a Data Management System with PostgreSQL/PostGIS and R*. Springer, 2014.

[30] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, pages 97–106. IEEE, 2002.

[31] Adi Weller, Yotam Orchan, Ran Nathan, Motti Charter, Anthony J. Weiss, and Sivan Toledo. Characterizing the accuracy of a self-synchronized reverse-GPS wildlife localization system. In *Proceeings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Vienna, Austria, April 2016. To appear.