Adaptive Pipelined Work Processing for GPS Trajectories

by

Andrew Hung Yao Tjia

B.Sc. Computer Science, The University of British Columbia, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

September 2012

Abstract

Adaptive pipelined work processing is a system paradigm that optimally processes trajectories created by GPS-enabled devices. Systems that execute GPS trajectory processing are often constrained at the client side by limitations of mobile devices such as processing power, energy usage, and network. The server must deal with non-uniform processing workloads and flash crowds generated by surges in popularity. We demonstrate that adaptive processing is a solution to these problems by building a trajectory processing system that uses adaptivity to respond to changing workloads and network conditions, and is fault tolerant. This benefits application designers, who design operations on data instead of manual system optimization and resource management. We evaluate our method by processing a dataset of snow sports trajectories and show that our method is extensible to other operators and other kinds of data.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

3G	3rd Generation mobile telecommunications technology
EDGE	Enhanced Data rates for GSM Evolution
ESHMM	Expanded State Hidden Markov Model
GigE	Gigabit Ethernet
GIS	Geographic Information System
GPS	Global Positioning System
GPX	GPS eXchange Format
HCR	Heading Change Rate
HMM	Hidden Markov Model
HSMM	Hidden semi-Markov Model
HSPA	High-Speed Packet Access
Mbps	Megabits per second
PDOP	Position Dilution of Precision
SDK	Software Development Kit
XML	eXtensible Markup Language

Acknowledgements

I would like to express my gratitude to both of my supervisors, Son Vuong and Eric Wohlstadter, for the opportunity to work with them, and their support and encouragement during my graduate studies. Through them, I was able to work on many interesting projects, and they allowed me the freedom to explore many different directions before finally arriving at this one. I would also like to thank Alan Wagner, who agreed to be my second reader, and for his invaluable feedback on this work.

My special thanks to all the members of the NIC lab, both past and present: Shahed Alam, Ricky Cheng, Stanley Chiu, Yong Chung, and Jonatan Schroeder. It was always a pleasure to discuss new ideas and get advice on my work. Their enthusiasm motivated, in a large way, my studies and research here at UBC.

Other people helped contribute to this work, and in particular, I would like to thank Byron Knoll for his guidance on implementing many of the statistical methods used in this work. My thanks also go to Daniel Lu, and especially Jackie Cheung, who read previous drafts of this dissertation, and provided many valuable comments that helped shape it to its current form today.

Most of all, I would like to thank my family. Without their tireless love and support over the years, none of this would have been possible.

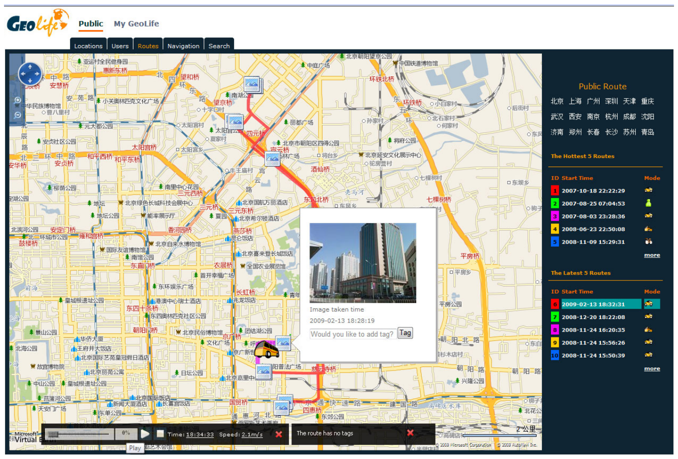
Chapter 1

Introduction

The relevancy of trajectory processing has been driven by the ubiquitous nature of GPS devices today. Portable handheld GPS units can capture trajectories in cases such an adventurer on a day hike, or a driver on a road trip. We have increasingly come to rely on geospatial information to help direct our lives; for example, geospatial information is embedded in many common forms of data such as images, microblog entries, and business databases. However, when evaluating trajectories, these applications fall short of deep analysis and are often limited to projecting a trajectory onto a map or as a log of one's location over a period of time.

A number of research projects, however, have extended much deeper processing and analytical techniques to trajectories. Geolife, a project from Microsoft Research Asia, collected trajectories created by people doing actions of everyday life, primarily around Beijing, and inferred their transportation mode, such as walking, bussing, or driving (Figure 1.1) [28–30]. Trajectories have also been studied especially in the context of querying and storage [4] and compression [24]. There is a wealth of postprocessing that can be applied to trajectories which we shall detail later on.

However, systems that have been designed to acquire and work with trajectories are often mobile devices such as smartphones, handheld GPS units and other embedded systems. This makes processing trajectories difficult due to the computational nature of trajectory processing. This systems challenge is also present at the other end where a shared server often resides. This is common in situations where trajectories are processed and stored "in the cloud" as part of larger systems involving geospatial queries or intelligence. *Adaptive processing* is a solution to this because by not hardcoding assumptions about the processing environment, *systems can perform better if they*



Public Route

北京 上海 广州 深圳 天津 重庆
武汉 西安 南京 杭州 成都 沈阳
济南 郑州 长春 长沙 苏州 青岛

The Hottest 5 Routes

ID	Start Time	Mode
1	2007-10-18 22:22:29	🚗
2	2007-08-25 07:04:53	🚶
3	2007-08-03 23:28:36	🚗
4	2008-06-23 22:50:08	🚗
5	2008-11-09 15:29:31	🚲

more

The Latest 5 Routes

ID	Start Time	Mode
6	2009-02-13 18:32:31	🚗
7	2008-12-20 18:22:08	🚗
8	2008-11-24 16:20:35	🚗
9	2008-11-24 15:56:26	🚗
10	2008-11-24 15:50:39	🚗

more

Image taken time
2009-02-13 18:28:19
Would you like to add tag? Tag

Time: 18:34:33 Speed: 2.1m/s

The route has no tags

Figure 1.1: In this screenshot taken from MSR's Geolife project, a trajectory is analyzed with mode of transportation and annotated with geotagged media. [29]

change their processing or algorithms in response to changes in environment.

Adaptive processing has been applied successfully in other domains to build systems tolerant of network delays and errors, low-power processing, and to mitigate other disadvantages that plague mobile devices [17]. Adaptive processing will be an important part in designing any comprehensive system that seeks to cover end-to-end processing of GPS trajectories and is the foundation of our system design.

We use adaptive processing to build a client-server architecture that uses *adaptive pipelined work processing* to perform several operations critical to GPS trajectory processing such as smoothing, activity recognition, and matching trajectories to geographic features representing real-world places from a Geographic Information System (GIS) database. The pipelined architecture makes the system resilient to faults and helps the system recover from errors. The system scales from light workloads to heavy workloads by changing the allocation of work between the client and server in the face of changing load and changing network conditions. With this framework in place, application designers can focus on designing the operations, instead of managing system details and resources.

1.1 Contributions of this Thesis

In this thesis, we investigate GPS trajectory processing by designing and implementing a system following the client-server architecture design paradigm, that adaptively and dynamically processes the trajectory. Although trajectory processing has been covered in earlier works such as [31] and variously in components such as in [28] or [15], to the best of our knowledge, there is no work addressing adaptive processing as relates to that faced by online processing of streaming GPS trajectories.

As part of our contributions, we discuss in detail the work that a trajectory processing system must handle. GPS trajectories have known errors due to environmental effects [19] which are corrected by smoothing. Activity recognition is a large field in its own right and we propose a simple, effective method for recognizing actions of the user as well. Another major processing step is to match geographic features to parts of the trajectory.

A second contribution is the exploration of constraints and requirements applicable in a trajectory processing system, and the proposal of adaptive processing algorithms to solve them. With adaptive processing we are able to overcome assumptions that do not hold in a mobile environment such strong connectivity and unlimited processing resources. By utilizing principles of adaptivity, we can delegate work to other processors to optimize some goal such as minimizing energy use or reducing processing time.

As a third contribution, we design a pipelined system using adaptive processing which can be extended to processing of arbitrary data. We evaluate it here using trajectory data acquired from snow sports participants and trajectory processing specific operations; however, the design is applicable to other scenarios involving pipelined operations over a network. We also demonstrate the feasibility and advantage of our adaptive system in a variety of scenarios. Based on this evaluation, we believe that this system addresses the needs of trajectory processing especially over mobile devices and networks. Finally, we propose future directions in which this work can be further extended.

1.2 Organization of this Thesis

First, we discuss related work in Chapter 2 like background algorithms and models that we use in GPS trajectory processing, and introduce other research in adaptive processing. However, trajectory processing occurs on devices that are much more limited in power compared to traditional computing architectures. To solve this, we introduce *adaptive pipelined work processing* in Chapter 3. Then in Chapter 4, we describe a typical processing algorithm for trajectory processing suitable for feedback generation.

We detail our implementation in Chapter 5. Chapter 6 describes our evaluation strategy where we pit our system against a variety of scenarios and discuss performance and fairness. Finally, Chapter 7 contains our conclusions and possible future work.

Chapter 2

Related Work

Adaptive processing in this thesis is motivated by the processing steps that are typically done to GPS trajectories. From the time of acquisition, a typical processing workflow may invoke some or all of the following steps such as smoothing, activity recognition, and matching to GIS features. These steps are done in order to discover patterns, and extract meaning from the trajectory. Other steps compensate for hardware or environmental errors. Taken as a whole, they form part of a larger system such as one that can answer queries on the data or predict future data. We shall implement these algorithms later in Chapter 5 as part of our processing system.

2.1 Smoothing

In a system that depends on sensor readings, we want to correct as much sensor error as possible. In our trajectory processing system, the error results from the noisy nature of the Global Positioning System. Though modern GPS units have facilities to correct error, these errors still persist due to the effects of the environment. GPS errors exist due to changes in signal from atmospheric effects, multipath signal propagation, and clock errors [19].

The process of removing error, or more formally, guessing the unknown variables from the observed measurements, is called *smoothing*. In the context of trajectory processing, this means determining the precise latitude and longitude, altitude and speed. A small amount of error is acceptable, but large amounts of error can mar the ability of subsequent steps to be performed accurately.

Our discussion of smoothing begins with mean and median smoothing. Then, we improve on this by describing the Discrete Kalman Filter [13].

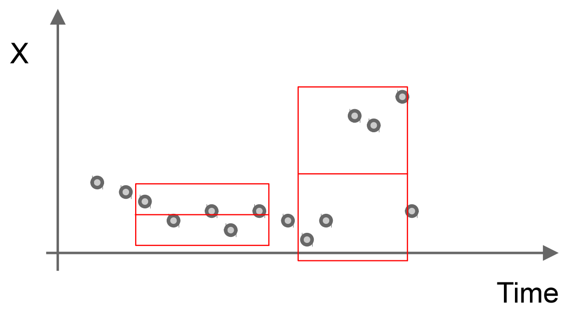


Figure 2.1: In this diagram depicting mean smoothing with a fixed window, measurements are taken representing the x displacement of an object over time. The red boxes represent windows that the algorithm computes the average over, which are represented by the midlines subdividing the boxes.

2.1.1 Mean smoothing

Mean/average smoothing is also known as calculating a moving average. By extending a window through a time sequence of data, and calculating the average of all points in that window, it is possible to obtain a smoothed estimate of some point that reduces the effect of noise on data. The operator will choose some window size, for example, $n = 2$. Then, for some measurement x_i, we can obtain the moving average estimate for x_i by calculating:

$$x_i = \frac{\sum\limits_{j=i-n}^{i+n} x_j}{2n+1}$$

The choice of window size is often determined empirically — that is, one increases the window size until some measure of eliminating noise is satisfied. Increasing the window size too large, however, will begin to reduce the resolution of the data so there is an inherent trade off between noise reduction and data reduction.

Calculating a moving average, however, is highly sensitive to outliers in the dataset. This can be solved by median smoothing, which is discussed in the next section.

2.1.2 Median smoothing

Median smoothing is a variation on moving average smoothing where instead of calculating the average of all measurements in a window, it computes the median of measurements. The median statistic in a dataset is not affected by outliers compared to the mean. Consequently, median smoothing is resilient to outliers in the measurements.

2.1.3 Drawbacks of Mean and Median smoothing

The only configurable parameter in mean and median smoothing is the window size. A larger window size is beneficial if the impulse is small in degree — that is, if the body being tracked experiences little acceleration. However, if a body experiences larger amounts of acceleration, then a larger window size may add error and reduce precision. Therefore, this single parameter is not flexible enough to represent the complexity and variation of different inputs experienced by the system.

2.1.4 Kalman smoothing

The Kalman Filter is one of a class of algorithms that solves the Observer Design Problem — a class of problems where we desire to predict internal states from the output of some enclosed system [2]. This is similar, but different to the standard Hidden Markov Model, which we describe later in Section 2.2.1, as we try to predict continuous states rather than discrete ones.

The Kalman Filter [13] allows one to incorporate a physical model into its estimation along with a measure of error. At each step, the filter updates its internal state with a prediction based on measurements and a result driven by the physical model. The measurements and model can vary depending on the application — Kalman filters have been used in economics, communications, and computer vision, among other areas. The physical model can describe the movement of an object in space (kinematics), the fluctuation of the price of a financial instrument (computational finance), or decoding of lossy data (signal processing). Here, we use it in navigation, as it has traditionally been used such as for guidance control systems.

This is based on a process model [2]:

$$x_k = Ax_{k-1} + Bu_k + w_{k-1}$$

where:

$$\begin{aligned} x_k &= \text{state vector of process at time } k \\ u_k &= \text{control input at time } k \\ w_{k-1} &= \text{existing process noise at previous time step } (k-1) \\ A &= \text{transition from state } k-1 \text{ to } k \\ B &= \text{maps control input to state } x \end{aligned}$$

The filter is a series of equations that are applied recursively at each time step to reach an estimate for an entire track. The equations are composed of several major components, which we describe from the original paper.

Prior to using the filter, we must specify several additional matrices. The H matrix specifies how a measurement we obtain from a sensor, z, maps to the state x. The Q matrix is the noise generated by the model itself.

We now have all the pieces needed to assemble the Kalman Filter. First, we estimate the initial state, $\hat{x}_0$, and initial state error covariance, P_0. Then, at each iteration $k > 0$, we measure z_k and estimate R_k, the measurement error, and obtain estimate of state $\hat{x}_k$ by executing [2]:

$$\begin{aligned}
&\text{Time Update:} \\
&\qquad \hat{x}_k^- = A_k \hat{x}_{k-1} + B u_k \\
&\qquad P_k^- = A_k P_{k-1} A_k^T + Q_k \\
&\text{Measurement Update:} \\
&\qquad K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \\
&\qquad \hat{x}_k = \hat{x}_k^- + K_k (z_k - H_k \hat{x}_k^-) \\
&\qquad P_k = (I - K_k H_k) P_k^-
\end{aligned}$$

For smoothing, the sequence $\hat{x}$ of estimated states along with the sequence P of estimate state errors can be assembled into a smoothed estimate.

The Kalman Filter's speed depends on the processor's ability to multiply large matrices. The two main determinants of the filter's complexity are the size of the state vector, x, and measurement vector, z, which have implications for the size of the remaining matrices. This means that an efficient matrix operations library or specialized hardware like GPUs can speed up this processing, and can make a difference if the trajectory to be smoothed is long. This has important implications for our adaptive processor later as we aim to optimize this on the client and server.

2.2 Activity Recognition

Activity recognition is a long established field with applications in many different areas. It is the process of determining activities or actions of some entity based on indirect observations collected by some sensor platform.

Activity recognition may take a range of inputs depending on the usecase. For example, a video camera's feed can be analyzed using computer vision techniques in order to determine the actions of a subject being recorded. Or, body worn sensors can sense 3D motion to determine whether the user is running or jumping. We can also recognize activities further removed from the user, such as mode of transportation. Analysis on GPS tracks can also label the mode of transportation [28], such as whether the user is taking a bus, or walking, or driving. Indeed, we can extrapolate this data even more as Zheng et al. do and determine whether the user is at home, at work, or on his commute [28].

While we can sometimes use reason and logic to determine actions, in many cases, it is difficult to create a set of rules that fully describe even the simplest of human activities. In other cases, it may be impossible to observe the outputs necessary to determine the action, and we may only be able to make indirect observations. This naturally leads us to consider the use probabilistic models for activity recognition, of which, the Hidden Markov Model is a prime specimen.

2.2.1 Hidden Markov Models

Hidden Markov Models (HMM) are an extension to discrete Markov Models that are popular for activity recognition [21]. In discrete Markov Models, a system is modelled as a set of states with edges between them. During some time sequence, the system may undergo several state transitions where the system will change its state along the edges defined. A typical problem in this domain is to determine the weight of these edges, from some observed sequence of state transitions, which are deemed *transition probabilities*.

However, in many circumstances, the actual states are not directly observable. For example, in the case of sequence alignment in bioinformatics, we do not know beforehand what genes are being expressed in a DNA sequence. We only observe the patterns of AT/GC basepairs that are inherent in a DNA sequence. The extension of this model, the Hidden Markov Model, has the ability to predict these hidden states. In this case, the model is governed additionally by *emission probabilities*, which describe the probability that a state outputs a given observation.

More formally, in an HMM, we have a sequence of observations:

$$O = O_0 O_1 O_2 ... O_T$$

and a model λ:

$$\lambda = (A, B, \pi)$$

where:

$A = \{a_{ij}\}$ [the state transition probability matrix]

$B = \{b_{jk}\}$ [the observation symbol probability distribution in state j]

$\pi = \{\pi_i\}$ [the initial state distribution]

We then try to discover the state sequence $Q = Q_0 Q_1 Q_2 ... Q_T$ that maximally describes the observations (that is, best explains the observations). This can be done using the Viterbi algorithm which finds Q [23].

Let $V_{0,k}$ be the probability of the most probable sequence of states ending with k that explains the first observation, O_0.

$$V_{0,k} = P(O_0|k) \cdot \pi_k$$

Then, for the rest of the sequence, $V_{t,k}$ is the probability of the most probable sequence of states ending with k that explains the sequence of states up to O_t.

$$V_{t,k} = P(O_t|k) \cdot \max_{x \in S}(a_{x,k} \cdot V_{t-1,x})$$

It is not unusual for the number of HMM states to balloon quickly into the hundreds when modelling complex phenomena. In addition, we may observe long sequences of observations depending on the measurements taken. This implies higher CPU and memory demands in order to run Viterbi and other similar algorithms as the size of S grows and the length of O increases.

2.3 Adaptive Processing

Adaptive processing is a paradigm, inspired by biology, which describes systems which are capable of responding to changes in environment or conditions. An adaptive system has an objective that it aims to minimize by controlling the strength of some action. The action is the system's response that modulates the objective [14]. Consequently, there is a correlation that is established between three components: the system structure, system behaviour, and its environment [14].

There are two components to such systems. Firstly, the system must have the means to identify stimuli, or changes in environment, that are important to its internal model of adaptivity. Some changes in environment will impact the system and its behaviour, whereas others may be more benign. Secondly, the system invokes *anticipatory behaviour*, which is any behaviour that is derived from its expectations about the future. These expectations are driven by the stimuli identified in the first part.

In order for us to consider adaptivity, we first need to decide which objectives we want to optimize in our processing pipeline. Our primary objective might be to optimize performance. In this scenario, we would like to have trajectories processed speedily from time of acquisition to when it is stored on a server. In this way, the data can then be available immediately for other applications server-side to consume and make use of.

However, there are several other objectives that could follow from our use of mobile devices. We might require processing to be robust and reliable; that is, it can withstand network outages and will not be corrupted on the wire. Another objective that is mobile device specific is to minimize use of mobile device system resources since the amount of resources on the mobile side may be much more constrained than is available on the server. This also has implications on energy usage.

Clearly there are some trade-offs to be made here. Adaptive processing might overcome limitations of mobile computing such that processing can still occur with deteriorated or even lack of a network connection, but at a performance cost. In another instance, shifting the processing burden from the client to the server may reduce the computational cost of the processing and consuming less device resources (e.g., CPU and energy), but might increase processing demands on the server. Furthermore, compared to performing operations client-side, the results of server-side performed operations may not be as readily accessible by the client. Alternatively, adaptive processing may also reduce system load when the server is in a shared environment with spiking loads, but at the expense of consuming more client system resources.

2.3.1 Mobile devices

GPS trajectories are often recorded on mobile devices. Mobile devices are a relatively new phenomenon that have become ubiquitous in recent years. Our car navigation systems, music players, and e-book readers are all representative of this segment. They are pervasive in sensors to monitor traffic flow, weather, or crowds in public areas. However, the statistic that most defines this segment are the 500 million smartphones that are sold worldwide every year.

A smartphone sold by Apple or Google such as those running the iOS or Android mobile operating system, typically have mobile data connectivity over technologies such as 3G/4G or HSPA (High-Speed Packet Access), in addition to normal cellular voice service. It is also common for smartphones to have a variety of sensors such as a GPS sensor, an accelerometer, or even a barometric sensor. Lastly, these devices are highly programmable compared to other embedded systems, making them an ideal target for exploration. However, there are pitfalls to using mobile devices, many of which are identified by Pentikousis [20]:

Unreliable network Mobile devices run on networks that are typically slower than commonplace household or business broadband installations. For example, wireless carriers often have HSPA widely deployed, which supports up to a 14.4 Mbps (megabits per second) download and

5.76 Mbps upload transfer rates in ideal environments [25]. However, when coverage is insufficient, speeds can degrade to EDGE speeds (Enhanced Data rates for GSM Evolution) around 50 kbps (kilobits per second) in both directions.

In addition, mobile networks are unevenly deployed throughout the environment, and wireless coverage may vary greatly depending on the user's location and time of day. Network connectivity can also degrade in crowded spaces with many users. Consequently, network access may inconsistent for device applications, as the user changes their location.

Constrained processing resources Mobile devices have weaker CPUs usually one or two generations behind their desktop counterparts. Furthermore, mobile devices often have very aggressive frequency scaling to achieve power management goals so the amount of processing resources available can vary widely for a single device. Mobile processors and hardware are often designed in lockstep with power consumption as they are closely related.

Limited power consumption While processing has followed Moore's Law to a certain extent, battery technology has not. Pentikousis estimates that battery technology has only increased 50 percent since 2002 [20]. Die shrinks and smaller process sizes of circuits have alleviated this somewhat in recent years, but this area is often compromised by increasingly complex hardware and processing requirements.

Furthermore, users according to Pentikousis are increasingly relying on their mobile devices to accomplish many different tasks, so a single battery must power a range of different computing tasks [20]. These tasks are also growing in complexity, scaling with hardware, consuming many processing cycles.

While these issues do not affect all mobile devices equally to the same degree, any system that incorporates mobile devices must anticipate and accommodate any or all of these problems.

2.3.2 Cloud computing

Cloud computing is an abstraction for access to a variety of computing resources over a network such as the Internet and is made up of many servers that power these resources. It is common to discount these servers as having unlimited resources. However, at the other end, these servers are physical computers which are subject to much of the same limitations as the computers found in our mobile devices, but at a different order of magnitude.

Furthermore, cloud servers often use shared machines dedicated to more than one task. These tasks may be related to the task at hand (such as `syslog` logging, or replication), or may be unrelated tasks (such as multiple applications on a single server). Therefore, an application designer may not be able to anticipate the amount of computing resources available at runtime. Secondly, cloud servers may also need to deal with varying processing demands which are not uniformly distributed. Therefore, even on dedicated machines, processing resources may not be sufficient to cope with all incoming requests.

One such way that cloud computing has solved these issues is by dynamic scaling. New application instances are provisioned instantly to meet demand as needed. While this has been used in some real world deployments, for the remainder of this thesis, we shall consider a more static model of cloud computing in which processing capabilities are fixed. Thus, the emphasis will be on processing efficiently given some upper bound on resources.

2.3.3 Implications for processing

At this point, we have established that both the client and server on both ends are subject to different constraints. However, they both lead to the same conclusion — at neither end can we establish a constant model of processing. The mobile client will be governed by weaker CPUs that are subject to aggressive frequency scaling and must bow to power management constraints. The server may or may not be shared, and experiences a non-uniform request rate when demand or popularity spikes.

As Mummert et al. conclude [17], mobile computing is best solved with adaptivity. We summarize the five problems of mobile computing that concern adaptive processing and data that is processed by clients and servers [17]. These issues are results of the mobile computing pitfalls identified in Section 2.3.1.

- Because of changing network conditions, data acquired on the client might not always be propagated to the server in a timely manner.
- Mobile computing may incur latency to fetch data needed for processing — a *cache miss*.
- In a mission critical environment, client data cached on mobile devices may be at risk, due to the security implications of losing the device or incurring damage.
- Where many updates exist on a shared resource, *update conflicts* may arise.
- Caches may fill up on a mobile device, causing updates to be lost.

2.3.4 Strategies for adaptiveness

Adaptiveness as a means for network independence

As devices such as mobile devices which use adaptive processing are user-facing, the integration of adaptive processing should aim to be relatively transparent. Systems such as Puppeteer [5] are designed to be transparent to network failures. This was accomplished to such a degree that integration with Microsoft PowerPoint was possible as an add-in to allow partial retrieval of slideshows on demand. An adaptive system promoting network independence should expose interfaces which hide adaptive details and allows operations to be implemented without knowledge of its network details. There should be a clean separation between application code, such as PowerPoint code, and adaptive processing code.

Adaptiveness as a means for increasing backend scalability

Adaptiveness should allow reallocation of work to allow the backend to dynamically adjust with processing load. There is much prior research in this area in the distributed systems field which addresses this issue using work queueing and other strategies such as work pushing or work stealing. However, our system is much simpler as we are designing around a single server and many clients.

One such paradigm to work processing is designing the system around a central pipeline or chain of modules. Chandrasekaran et al. puts forth a compelling case for decomposing complex services as a set of simple services working together in *Ninja* [3]. While performance may not exceed that of purpose built systems, modular systems can allow us for more robust failure recovery and adaptability [3]. This is true in a wide range of applications such as building robust web sites. Websites now must deal with more complex data-driven pages requiring extensive processing or heavy media assets and also must scale to accommodate a massive influx of visitors brought by content aggregators such as Reddit or Slashdot without interruption. No single system can scale infinitely; at some level, the constraints of hardware come in. Another issue that is important in high load situations might be fairness — or how evenly the performance deteriorates for all users. SEDA [27] redesigns applications to use event-driven stages fed by queues that scales fairly as usage increases — that is, the overall performance decreases equally for all users as the load increases, rather than a few choice requests being answered and the rest rejected.

This design is evident in other applications such as building efficient network stacks or in live video encoding [16]. In either of these cases, it may be desirable to minimize latency. In the case of a network stack, the round trip time of a packet is important, and in live video encoding, processing delays may make the live stream diverge from the processed stream. In Scout, each module is given information about its processing context which allows each module to make runtime decisions on its processing [16]. For example, in the case of high load, a video streaming system may decide to reduce the quality of the resultant video. Another case might be a networked system that selectively drop incoming packets so as not to reduce the quality of service according to some rule.

Pipelines may also have other advantages such as having more robust failure recovery and fault isolation [16]. Due to the fact that the system has less “moving parts” and less complex dependencies, it is easier to identify where faults occur, and by extension, easier to recover from these faults.

Chapter 3

Adaptive Pipelined Work Processing

The operations we will introduce in Chapter 4 are essential to the processing of many types of GPS trajectories. Though we have pointed out limitations of GPS trajectory acquisition hardware and some examples of adaptive systems in Section 2.3, we have not yet reconciled the problem of how to execute these operations optimally given the constraints.

The solution is a client-server system using *adaptive pipelined work processing*. It can adapt to a variety of different processing steps, workloads, and hardware configurations.

3.1 Building an Adaptive Pipeline

An adaptive pipeline is formed of *operators* which are executed by one or more *processors* and specified by a *processing plan*. The adaptivity we introduce in this thesis comes from how to execute a processing plan over several processors. In addition, operators tailor specific work from the *operating context* provided by the processor.

3.1.1 Pipeline components

Operands

Operands are the units of data in the pipeline. They represent all input data and all results. The input to our adaptive processing pipeline shall be a trajectory wrapped within an operand.

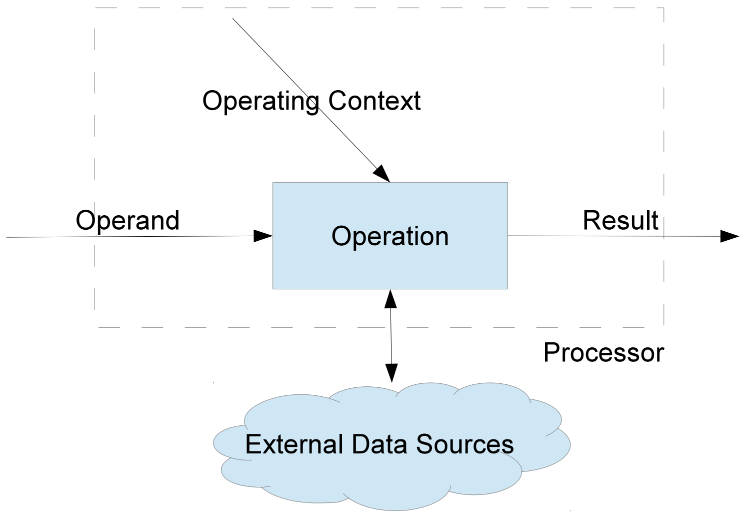


Figure 3.1: In this figure of a sample pipeline stage, an operator receives input data in the form of *operands* and returns a *result*. The operator is provided an *operating context* by the processor.

Operands must be distinctly serializable to memory at any point in the pipeline workflow. This allows it to be stored temporarily, queued in a work queue, or transferred across the network if called for by the processor. As a consequence, operands must also ideally provide an estimate of their serialized size to avoid expensive serialization routines, in order to allow the algorithm to anticipate network transfer times and storage requirements.

Operators

We now introduce *operators*. An operator takes *operands* which are arguments similar to function arguments in a programming language, and performs some operation necessary for our system. It also returns a result which can then be passed into further operators. Operators are provided with the *operating context* which allows the operation to be aware of the current state of the system and its execution environment.

Operating Context

The *operating context* is information about the environment that an operation is running in, and is inspired by Scout [16]. The environment may vary because of specific hardware and architectures which have different computing requirements. These static parameters may usually be specified ahead of time, since hardware or architecture is unlikely to change in the short run. In addition, dynamic parameters can be calculated such as the load average or the available amount of free memory, which may affect processing decisions. On a mobile device, other dynamic parameters could be the current charge level of the battery or even the user's location.

Processing Plan

For each trajectory, a *processing plan* is individually created depending on the trajectory. For example, if the trajectory delineates a road trip, a processing plan may run operators that identify which streets were driven on, estimated traffic congestion, or even calculating the carbon offset required to compensate for the journey. However, if the trajectory represents a skier on a mountain run, then a plan's operators may recognize ski activities, or match the trajectory to ski runs. Each action should be represented by an operator and is assembled into a processing plan.

Let the processing plan P consist of n operators such that

$$P = [O_1, O_2, O_3 ... O_n]$$

We use a simplified model of a linear pipeline such that each operator is executed sequentially. However, we note that this need not necessarily be the case — operators may not necessarily be dependent on the operator immediately preceding it and a future optimization might be to execute operators in parallel when possible.

Processor

A *processor* is a component of both the client and server which is responsible for applying operators to the data as described by the processing plan and providing the operating context as necessary. The processor integrates the components described in this section and fulfils the processing plan or delegates work to another processor (such as the client pushing work to the server).

3.2 Adaptive Processing Decisions

From the components we have defined in the previous section, an adaptive processing algorithm must decide the assignment of operators to processors. Figure 3.2 illustrates how the client and server coordinate their processing decisions.

3.2.1 Processing heuristics

In order to make decisions on how to allocate processing, processors record the duration of past operations. This also gives a distribution of the complexity of the operation with respect to the input data. The length of time is stored in a continuously updating histogram H; a unique one which is built for each operator. The algorithm maintains a collection of separate histograms for the client and server, H^c and H^s, such that:

$$H_i^x = \text{Histogram for } O_i \text{ running on } x$$

In addition to supporting "insert value," each histogram can report an approximate value at a given percentile. Therefore, we can estimate the value at the 50th percentile on histogram H by invoking:

$$valueAtPercentile(H, 50)$$

We can also similarly get the percentile for a value instead. To retrieve the approximate percentile that the value, 10, falls in for histogram H, we

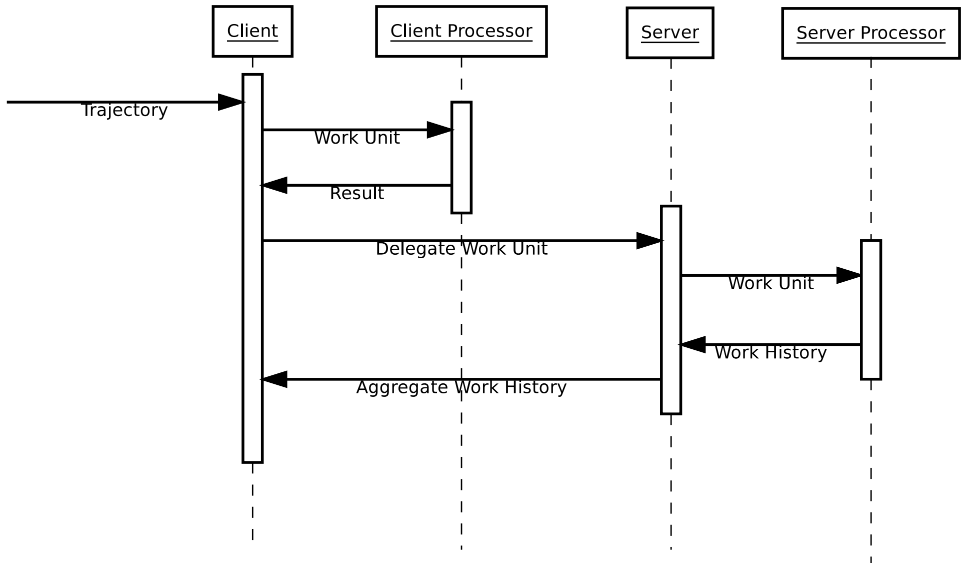


Figure 3.2: This sequence diagram shows the interaction of the client and server processors. When a trajectory is received on the client, a *work unit* consisting of the trajectory *operand* and *processing plan* is submitted to the processor (Section 3.1.1). The processor will process a work unit until the client decides to delegate (Section 3.2.4). When the work unit is delegated, it is sent to the server and processing is resumed there. The server will send back a processing history object containing the details of the past executions that is used by the processing heuristics calculations (Section 3.2.1).

invoke:

$$percentileForValue(H, 10)$$

Furthermore, the client and server maintain moving average measurements of their estimated load percentile (Section 2.1.1). This is done by querying the running time after an operation has been completed against historical data in the histogram, and retrieving the percentile. Hence, the following definitions cover the estimates of load percentile:

$$estimatedClientPercentile = \text{Client's estimate of load percentile}$$
$$estimatedServerPercentile = \text{Server's estimate of load percentile}$$

Each client can also configure a priority parameter which is a scalar that modifies its own estimate of running time, P_c. We can estimate the time taken to execute operation O_i on the client, $T_c(O_i)$ choosing the value at the client histogram, H_i at the estimate of client load percentile.

$$T_c(O_i) = P_c \cdot valueAtPercentile(H_i^c, estimatedClientPercentile)$$

P_c defaults to 1.0, but by changing this, we can influence the adaptive processing decisions taken by our model such that operations on the client look cheaper than running on the server, or vice versa. We obtain a similar measurement for the server, $T_s(O_i)$, which is not scaled at all.

$$T_s(O_i) = valueAtPercentile(H_i^s, estimatedServerPercentile)$$

3.2.2 Operator selectivity

Many operators manipulate data in some form during processing. In the case of smoothing, data might be simply changed. For activity recognition, additional data is generated by the tagging of activities to the trajectory. When trajectories are compressed, the output of the operator is presumably less than the input operands.

While it is not possible to know ahead of time the precise amount of data generated or lost, it is reasonable to approximate it by calculating the *selectivity ratio*. For some data, x, the selectivity ratio, R, is the ratio of output data to input data, where $sizeof(x)$ is the size of x in bytes.

$$R(O_i, x) = \frac{sizeof(O_i(x))}{sizeof(x)}$$

Suppose we store all past observed ratios in a unique histogram for each operator and the data size for all observed data.

$$H_i^s = \text{Histogram of selectivity ratios for operator } i$$
$$H^d = \text{Histogram of data sizes}$$

Then, for an unprocessed data x we can estimate the selectivity ratio:

$$\hat{R}(O_i, x) = valueAtPercentile(H_i^s, percentileForValue(H^d, sizeof(x)))$$

3.2.3 Network transport

Network transmission is comprised of several parts. First the data must be serialized into the protocol format. Once the data is ready to be sent over the network, the network protocol may necessitate some kind of handshaking between the communication ends. Next, there is the actual network transfer over the link, and some additional overhead which may be inherent in the protocol. Finally, there is an acknowledgement and sometimes, a protocol teardown. Each stage has delays inherent within it such as processing, queueing, and propagation delays and is dependent on medium, hardware, system load and other conditions.

To make it easier for our model to estimate the network transport cost, we consider the entire network flow as a single unit and measure the total time it takes for data to be transferred between nodes. In doing so, we ignore all the component delays in a network stack (such as transmission, queueing, propagation, etc.). The amount of data transferred is measured, and the rate of *bytes/second* is measured from a simple linear cost model. A decaying moving average is updated with past transfers to estimate current transfers.

Therefore, the time in seconds to transfer some work is calculated by $U(bytes)$ where $bytes$ is the size of the work in bytes.

$$U(bytes) = \frac{bytes}{networkRate}$$

3.2.4 Delegating work

Determining when to delegate means we must choose the set of operations executed by the client, C, and the set of operations executed by the server, S by determining m such that

$$\begin{aligned} C_m &= [O_i | 1 \leq i < m] \\ S_m &= [O_i | m \leq i \leq n] \end{aligned}$$

To choose the optimal path for data x, we minimize m with $O(n)$ time complexity, where n is the number of operators:

$$\begin{aligned} \arg\min_{\mathbf{m}} \Bigg[\Big(\sum_{O_i \in C_m} T_c(O_i) \Big) + \Big(\sum_{O_i \in S_m} T_s(O_i) \Big) + \\ U\Big(sizeof(x) \times \prod_{O_i \in C_m} \hat{R}(O_i, x) \Big) \Bigg] \end{aligned}$$

Every time an operator is completed, the processor re-evaluates the optimal path by finding m for the remaining operations (Figure 3.3).

3.2.5 Recovering from faults

Recovering from failures is facilitated by this pipelined design. We consider that faults may happen in individual operators from some processing error (i.e., low memory, electrical fault, etc.). In this case, the operands are captured before the operator is executed, and the operator can be re-executed as necessary with a random backoff delay until the fault is corrected.

As well, faults may occur due to network such as loss of connectivity or high packet loss. In this case, network transfers can be retried as necessary.

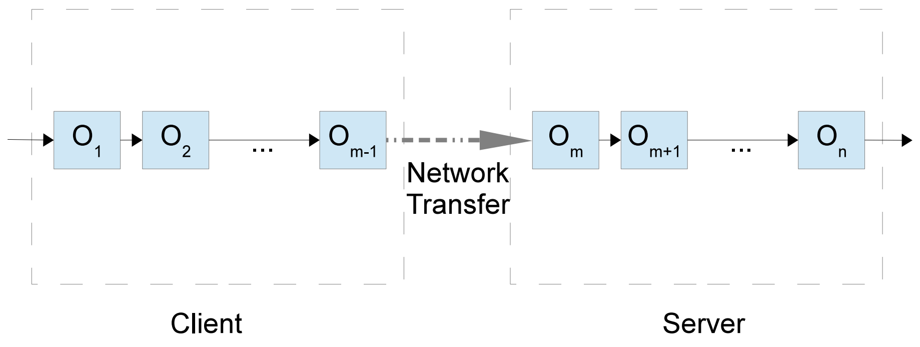


Figure 3.3: This figure illustrates the client/server split of the processing pipeline. Operations O_1 to O_n are split linearly at index m. At this point, the result from O_{m-1} on the client is transferred to the server, which resumes processing at O_m. The adaptive processor chooses m to optimize work delegation such that the total time taken processing on *both* the client and server plus network transfer time is minimized.

If network outage is prolonged, operands can be serialized and resumed when network connectivity is restored.

While we claim our system is fault tolerant, we do not explicitly test this capability in our evaluation, and choose instead to focus on the performance. Our confidence for the fault tolerant capabilities of our system stems from the large amount of existing systems that use this paradigm. For example, Internet email messages are queued and sent via SMTP, and often a single mail message may require relaying by multiple SMTP servers. If the receiving server is unavailable during any leg of an email message's journey, messages are queued and delivered at a later time. The pipelined design limits errors to individual pipeline stages and prevents errors from propagating if errors do occur.

Chapter 4

Processing Operators for GPS Trajectories

Postprocessing GPS trajectories is expensive computationally, but necessary in order to extract meaning and inference from the data. We begin with a discussion of our assumptions about the input data, then enumerate and briefly explain each technique that we use. Figure 4.1 summarizes at a high level the major operations that we cover in this section. Finally, we close with our scheme for adaptive processing.

4.1 Data format

The data is a time series of data points which are measured at mostly uniform time intervals (although we do allow for non-uniformity, such as when there is a sensor platform outage, or similar). Each data point should at minimum have a GPS measurement associated with it. A proper GPS position fix reports back several data components such as the 2D position, in latitude and longitude coordinates. With data from a sufficient number of GPS satellites, a 3D position fix can reveal the altitude and it can also report speed.

- Latitude and Longitude
- Altitude
- Speed

To store trajectory data in an interchangeable manner, we use the GPS eXchange Format (GPX) which is widely supported by different applications.

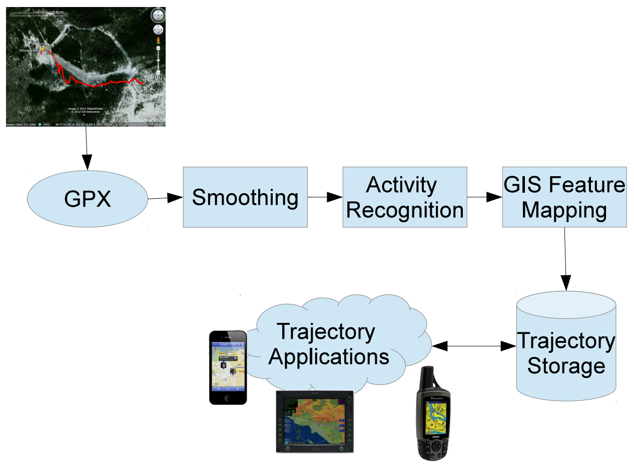


Figure 4.1: Trajectory processing overview

High Level Operation	Functional Description	Location
Trajectory Acquisition	Acquisition of trajectory from GPS hardware. Data acquired is specified in format detailed in Section 4.1.	Client only
Smoothing	Elimination of errors in trajectory. Section 4.2.	Client/Server
Activity Recognition	Match actions and intent to trajectory. Section 4.3.	Client/Server
GIS Feature Matching/Regionization	Align geographic features to trajectory. Section 4.4.	Client/Server
Trajectory Compression	Compress trajectories to save space. Section 4.5.	Client/Server
Persistence	Storage of trajectory and analyzed data to storage	Server only

Table 4.1: Trajectory processing operations

GPX stores this tuple using XML (eXtensible Markup Language) and is extensible to hold additional metadata.

4.2 Smoothing

As rationalized in Section 2.1, smoothing is necessary to compensate for error in any kind of sensor platform, such as a GPS sensor. We correct errors in trajectories by use of the Discrete Kalman Filter. The usage of the Kalman filter requires us to make several choices based on our application of estimating GPS tracks. First of all, we choose the state of the Kalman filter model with a suitable 4-dimensional state space:

$$x_k = \begin{bmatrix} x \\ y \\ dx \\ dy \end{bmatrix}$$

The x and y values correspond to the longitude and latitude values respectively in decimal degrees. The dx and dy values are the velocity vector components in the x and y axes of the object's motion. Once we have decided on the state space, we then decide on the model dynamics that describes the transfer of state space over time. Based on the state space, we set up the filter using a 2D kinematics model describing a particle through space.

$$d_{i+1} = v_i(t_{i+1} - t_i) + d_i$$

That is, the displacement in one dimension, d_{i+1}, at the next time step is the previous displacement, d_i, adjusted by the difference in time elapsed, $t_{i+1} - t_i$, between two measurements multiplied by the velocity, v_i. This is represented by the transition matrix, A, with $dt = t_{i+1} - t_i$:

$$A = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We cannot observe control inputs in this scenario as we have no means of ascertaining, for example, whether in a car's trajectory, a steering wheel was turned to the left or right. Therefore, the matrix, B, and control vector, u, are set to 0. Each measurement, z, is composed of the longitude and latitude in decimal degrees, as well as the velocity components, dx and dy, which is measured from the speed and the GPS heading.

$$z = \begin{bmatrix} x \\ y \\ dx \\ dy \end{bmatrix}$$

As the order of the measurement components and the state components is the same, the H matrix is the identity matrix:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Lastly, we need a way to measure the error of measurement z, represented by the noise covariance matrix, R_k. One of the chief difficulties of using Kalman filter techniques is specifying an estimate for this input, since it is not always the case that we have sensor measurements with accurate measures of error. However, most consumer GPS units report an internal measure of error: the position dilution of precision (PDOP), which is the magnitude of the user position error (1.0 is a perfect measurement, whereas around 3.0 is typical for an “accurate” position). From this, it is possible to calculate the standard deviation of the receiver position, σ_{rc} [19]. In general:

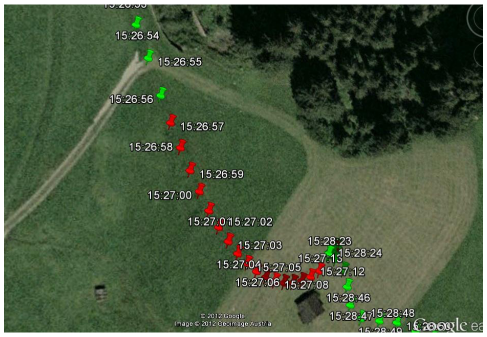


Figure 4.2: In this screenshot taken from Google Earth, a section of a trajectory is projected onto a map with a discontinuity caused by GPS error. The green pushpins represent original points taken from the hardware, but the red pushpins denote points where the original points were either not available or discarded due to high error, and are predicted instead by the Kalman smoother.

$$\sigma_{rc} = \sqrt[2]{\mathrm{PDOP}^2 \times 6.7^2 + 1^2}$$

From this, we can construct the diagonal matrix, R, which is the noise covariance matrix of the measurement.

$$R_k = \begin{bmatrix} \sigma_{rc} & 0 & 0 & 0 \\ 0 & \sigma_{rc} & 0 & 0 \\ 0 & 0 & \sigma_{rc} & 0 \\ 0 & 0 & 0 & \sigma_{rc} \end{bmatrix}$$

We have described one possible set of components that can utilize the Discrete Kalman Filter to solve the trajectory smoothing problem. In the context of an adaptive pipeline, these matrix operations can be performed on either the client or server side. A larger state space may be able to model more complex behaviour, but would incur greater processing demands.

4.3 Activity Recognition

Many human activities such as travelling to different locations or participating in a sport produce GPS trajectories. We employ activity recognition to discover the original activity from the trajectory. In this section, we demonstrate the derivation of indirect measurements from the original sensor data and construct a Hidden Markov Model to recognize a specific set of activities.

4.3.1 What can be gleaned from GPS tracks?

As the Geolife dataset showed, it is possible to determine the mode of transportation from GPS tracks. This is because each mode of transportation is distinct in terms of how movement occurs, in which direction, etc. The GPS track provides several information sources which is a tuple consisting of latitude and longitude, altitude, and speed (see Section 4.1.

From this track, we can calculate:

- Velocity
- Acceleration
- Heading
- Vertical speed
- Vertical acceleration
- Heading change rate

The velocity is simply the slope or the first derivative of the displacement in latitude and longitude, and the acceleration is the second derivative. Heading is obtained by taking the angle of the velocity vector with respect to North. The vertical speed and acceleration are derived from the altitude and its derivatives. The heading change rate is calculated as the rate of change in heading over time [28].

4.3.2 Constructing a Hidden Markov Model for Activity Recognition

There are many of ways of constructing the HMM state space as well as the transition and emission probabilities. We shall describe one such scheme here which produces good results for our usecase.

HMM structures can vary from ergodic (fully connected) to Bakis/left-right (in-order states) constructions. We use elements of both designs — while we allow the HMM to be ergodic in structure, we bias it to follow Bakis semantics through weighing transition probabilities in an ordered sequence. In doing so, we can express beliefs such as "Activity 2 often follows Activity 1 and takes about 10 minutes to completion."

The HMM is constructed to model duration, as activities have a real sense of a period of time over which it happens. There are many such strategies in which to model duration in an HMM such as Hidden semi-Markov Models (HSMM) [18]. We use a simpler scheme of Expanded State Hidden Markov Models (ESHMM) [12]. These are demonstrated by Johnson to have comparable performance to an HSMM and other schemes with much lower computation and complexity necessary [12].

In an ESHMM construction, there are a variety of ways to model duration, all of which rely on adding additional states to the HMM. We use a variant of a Fergusson HMM topology where each state is connected to the next and linked with a jump arrow to the next state (Figure 4.3) [12]. In Figure 4.3, the lower curved arrows transitioning from “Activity 1” to “Activity 2” are weighed with low probabilities, compared to the horizontal inter-state arrows within “Activity 1”. The Viterbi algorithm finds the most likely state sequence that explains the observations. In this case, Viterbi determines the most likely sequence of activities with regard to the duration modelling inherent in this construction.

First, the trajectory is transformed into our observation sequence such that each observation is ordered by time, and contains positional data, velocity, and other data extracted in Section 4.3.1.

Secondly, we must identify which activities are desirable to identify from

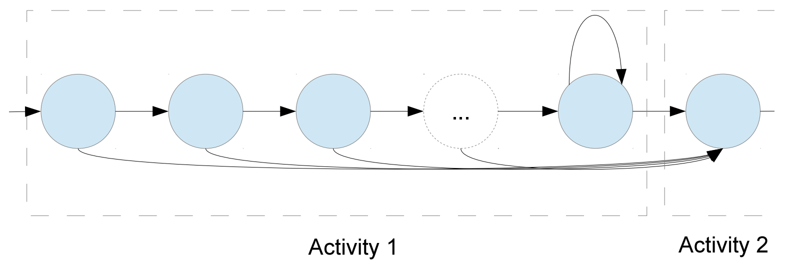


Figure 4.3: Modified Fergusson HMM topology for activity recognition [12]

the trajectory. For example, in a set of trajectories recording snow sports, we would like to classify different activities relevant to snow sports, such as when the user is skiing on a trail, on a lift, or merely walking around. Each activity has certain attributes. On average, we might reasonably conclude that on a ski trail, the vertical speed should be negative (due to the downwards slope), and the opposite should be true for a lift state. People might walk and loiter with an average speed of 0 to 5 km/h, but ski with speeds in excess of 20 km/h. We also use the heading change rate (HCR). The HCR should be highest when walking. However, when skiing on a trail, HCR is lower due to the lower manoeuvrability afforded by ski gear. It is lowest when on a lift since lifts are often constructed in straight lines between two points. In this way, we manually choose features based on our own knowledge of the world, but another alternative might be to learn the features automatically.

To construct our HMM, for each activity, we construct a sequence of states constructed in the Fergusson manner above, with each state's emission probability as a function of the manually identified features. Each sequence of states representing an activity has a "jump" arrow to the start of all other states, weighed by the activities being modelled. For example, it was observed in Geolife that walking connects all transportation modes together such as when an individual changes from driving a car to catching a bus [28]. In a snow sports dataset, it is reasonable to assume that there is some walking or loitering around that connects different activities together.

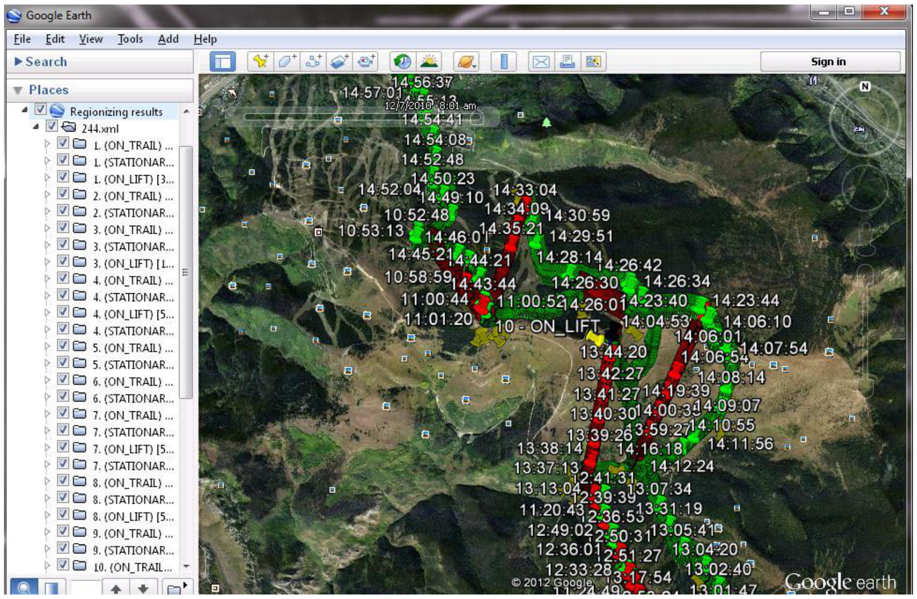


Figure 4.4: In this screenshot taken from Google Earth, a trajectory is projected onto a map with different coloured pushpins denoting different activities. The green pushpins represent the algorithm's best determination when the user was skiing on a slope, whereas the red and blue pushpins denote other actions such as sitting on a lift or waiting around.

The activities are matched by finding the most likely state sequence that explains the observations with the Viterbi algorithm. We may further post-process this sequence such as merging short sequences, etc.

4.4 Matching trajectories with geographic features

Most trajectories occur in settings where we may have foreknowledge of the environment. Such is the case in geographic information systems (GIS) where we have databases of existing spatial or geographic features. These features are natural or man-made, and come in a variety of different shapes and sizes. For example, we may know of a description of a major Interstate freeway or a popular bicycle path. This is usually specified as a series of points forming a line string geometry. Alternatively, we might know of a nearby ski resort;

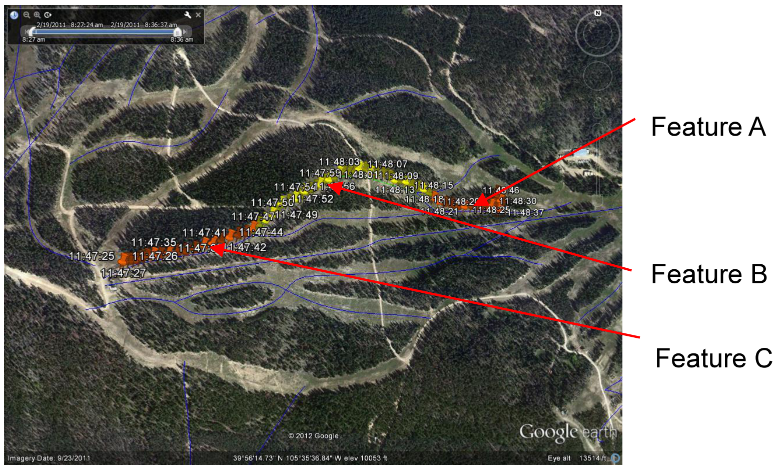


Figure 4.5: In this screenshot taken from Google Earth, a trajectory is projected onto a map with different coloured pushpins denoting different features. The GIS features are extracted from a known source, such as geospatial database, and can denote real world concepts such as roads, pathways, landmarks, regions, etc.

the entire bounds of which can be described as a rectangle or a polygon. Another common type of feature encountered are single points such as a landmark or an attraction.

Trajectories often interact with these features, and the goal of this section will be to establish a correspondence between elements of the trajectory and these features. For example, a driver in a car might want to know which road he is travelling on, to satisfy the goal of reaching a destination. Or, a plane might want to know which controller to contact based on their current flight path.

There are many approaches one can take to solve this problem. A simple algorithm might be to take the midpoint of the trajectory and compare it to the feature with the closest spatial distance — an operation that can easily be computed using a spatial database and R-tree indexes. One pitfall of this strategy is that it would only match a single feature for each trajectory. Additionally, the feature may not align well with the trajectory even though

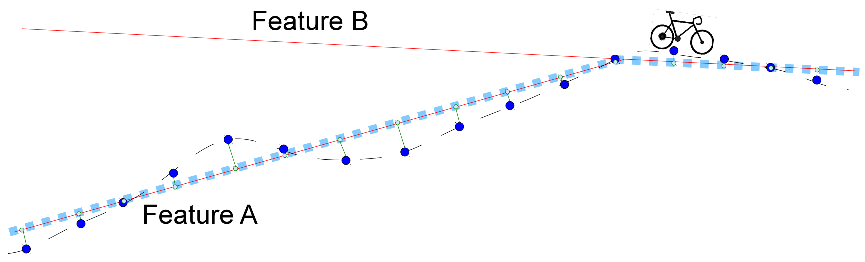


Figure 4.6: In this example, we have two features, A and B which intersect. The user's captured trajectory is denoted by the dashed black line, and the individual blue points indicate the observations that the HMM consumes. Each feature corresponds with an HMM state that has an emission probability that is a function of the minimum Cartesian distance between the observation and the feature (represented by the green "spokes"). The Viterbi algorithm decides the most likely feature assignment to each state and to the overall trajectory as represented by the dotted light blue line.

it is spatially close to the midpoint. However, we now discuss a flexible strategy that matches features probabilistically, which can be extended to many different trajectories and sets of features.

4.4.1 Using Hidden Markov Models to match features

We have already given an introduction to the Hidden Markov Model in Section 2.2.1 so we will now discuss how to construct the model to solve the problem of matching geographic features to trajectories.

Firstly, all relevant features that are considered for matches should be retrieved from the set of all features. Normally, such a set should be contained in a geospatial database. Then, all features can be obtained by a query such as retrieving all features that intersect some fixed size buffer around the trajectory. This dataset can further be pruned by only retrieving relevant feature types (i.e., only line strings, or those with a low Hausdorff or Fréchet distance).

Then, for each considered feature, we create a state in our model that corresponds to that feature. For a given observation, the state's output probability is a function of the Cartesian distance from the observation to the nearest point on the feature. Transition probabilities between states are functions of the Cartesian distance between the nearest points of the two features.

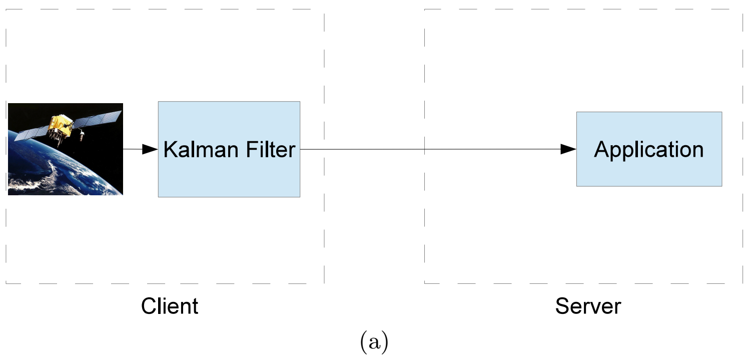
The Viterbi algorithm will decide the most likely state sequence, or most likely sequence of features, which matches the observed trajectory.

4.5 Trajectory Compression

Trajectory compression can be considered as the inverse of the smoothing problem described in Section 4.2. For example, smoothing methods such as Kalman filtering have the ability to "fill in the blanks" for missing data and compensate for a noisy sensor platform to great effect. The quality of this data is acceptable enough that evaluation of the dataset may show no discernible difference in accuracy versus one that was complete.

Jain et al. [10] use Kalman Filters to stream data across a network while minimizing data transferred. A *Dual Kalman Filter* architecture is used, by running Kalman filters both at the server and client (Figure 4.7). Where the Kalman prediction greatly varies from the observed measurement, the measurement is transferred. However if the error is small, the measurement update is omitted. In effect, we used cached Kalman filter parameters to estimate future measurements.

This scheme can be extended toward the goal of compression by storing only propagated updates from the client to the server. There is a configurable level of error which is a trade-off between storage efficiency and precision. The higher the acceptable error, the less updates would be propagated from the client to the server thereby saving bandwidth in the network scenario, or storage in the compression scenario.



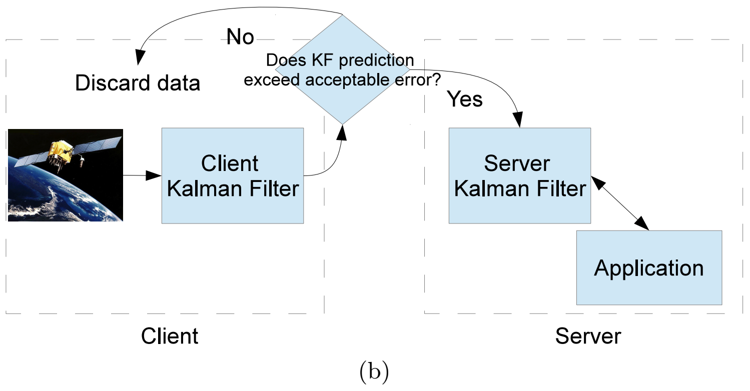


Figure 4.7: Dual Kalman Filter architecture. In (a), there is no use of a dual Kalman Filter, and so all updates are pushed from the server to the client. However, (b) illustrates what can be achieved by running Kalman Filters on both the client and server. A decision can be made whether to push the update or not, based on whether the KF prediction exceeds a certain configured error threshold. If updates are discarded, data is conserved.

Chapter 5

Design and Implementation

The design of our system using *adaptive pipelined work processing* spans a client and server architecture running on Android devices and cloud servers. It ameliorates performance and mitigates the pitfalls that are inherent in each component through the use of adaptive processing.

5.1 Hardware Platform

Recon Instruments produces the MOD Live — a heads-up display designed for snow sports [9]. It is a basic sensor platform incorporating GPS, accelerometer, and temperature sensors that reports this information in real time to the user (Figure 5.1).

It is programmable via a supplied software development (SDK) kit from Recon. It is also built on top of Google's mobile operating system, Android 2.3 Gingerbread, through which it can be programmed as well.

The MOD Live implements basic features that allow an athlete to receive feedback on his performance in snow sports. This feedback is often directly correlated to a sensor — for example, the MOD Live allows the athlete to view his current speed as reported by the GPS receiver. In addition, the MOD Live records this data into a RIB file (Recon Instruments Binary file format), which can be further analyzed by an accompanying application as well as a web community called the "HQ".

We have written software to interoperate with the MOD Live as well as work on the standard Android platform. Thus, our system can run on both a MOD Live and a generic Android device with the necessary sensing hardware.

Feature	Component
Processing	ARM Cortex A8 Core 600 MHz
Memory	512 MB
RAM	256 MB
OS	Android 2.3
Sensing	• Global Positioning System (GPS) Receiver • Tri-axial Gyroscope • Tri-axial Accelerometer • 3 Axis compass sensor • Temperature sensor • Barometric pressure sensor

Figure 5.1: Recon MOD Live specifications

5.2 Software Platform

We envision our system to not only be adaptive, but to work on real world hardware and situations. Therefore, the algorithms we have developed in prior sections must work well on mobile devices and backend servers. This is especially true in the adaptive processing scenario, as we would like to evaluate the effect of dividing up the processing pipeline between a client and a server backend. The Java Programming Language is well suited for this application, being available on a wide variety of devices, architectures, and operating systems. Its stated goal of "write once, run anywhere", while not true for many cases, does mean that if we adhere to certain programming principles and restrict the use of our API, we can easily port our code from one platform to another.

The end result is 9973 non commenting source statements (NCSS) of which the main adaptive pipeline and operators use 6874 NCSS and the web and mobile application code use 964 and 2134 NCSS respectively. Statistics and matrix computations are handled with libraries such as Apache *commons-math* [6] and the *efficient-java-matrix-library* [1].

5.2.1 Server

On the server side, we use a standard Java web stack built around the Spring Framework [26]. Spring lets us build the pipeline as a service-oriented architecture, which facilitates the swapping in and out of functionality through principles of dependency injection and inversion of control.

The server implements the pipeline processor endpoint as a WebSocket endpoint via the Bayeux protocol [22]. This allows for platform independence as the protocol is not system specific. Consequently, components can be implemented in different languages and communications libraries are widely available on mobile and desktop platforms. Furthermore, it is more efficient compared to standard HTTP methods and allows for bi-directional and full-duplex communications which cannot be achieved with HTTP methods without use of strategies such as long polling.

The server uses CometD as the Bayeux implementation. Eclipse Jetty

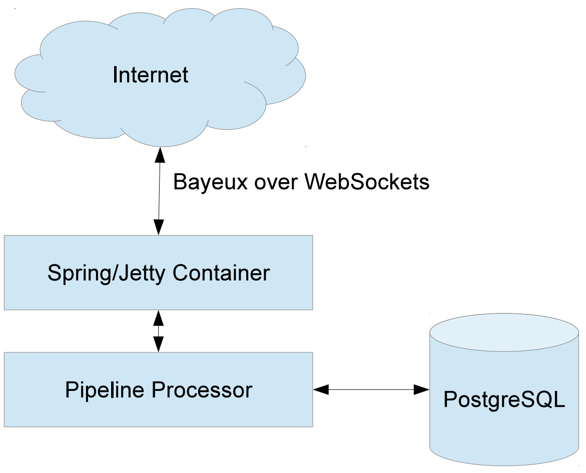


Figure 5.2: Server architecture

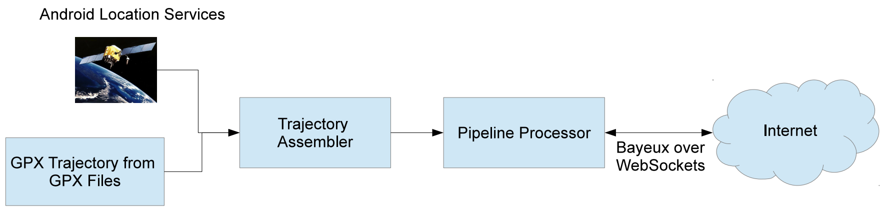


Figure 5.3: Client architecture

provides the servlet container and WebSocket server library. Relational data storage uses the PostgreSQL database management system and PostGIS spatial database extensions for operations such as trajectory storage and geographic feature retrieval.

5.2.2 Client

The adaptive processing pipeline is implemented on the client side via the Android SDK. Operators, being implemented in Java, are normally compiled into Java bytecode. Preparing operators to be run on Android requires cross-compilation from Java bytecode into Dalvik bytecode which can be executed

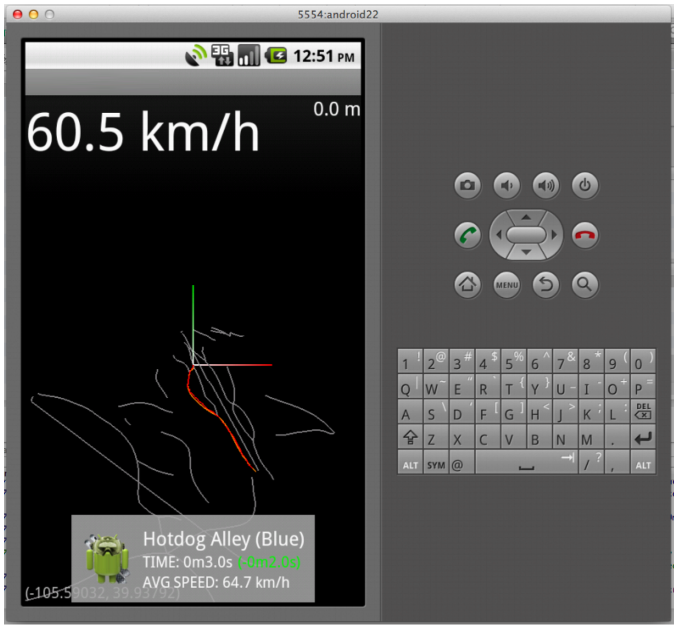


Figure 5.4: This screenshot, taken from the Android emulator depicts the prototype mobile client running on a simulated trajectory. The red track shows the trajectory recorded so far, which is the result of activity recognition detecting a new ski run, while other UI elements tell the user the current GPS parameters. On the bottom, a popup informs the user of the current feature that has been matched using the GIS feature matching operation.

on a variety of Android consumer devices.

Location data is obtained either from device sensors with use of Android location services or is read from serialized GPX files. The client also uses CometD client code to connect to the server endpoints (Figure 5.3).

5.2.3 Adaptive pipelined architecture

Pipeline operators

As part of an internship, we had previously designed our GPS processing system as part of a three-tiered architecture for web applications. That is, most of the processing happens on a logic and data tier on the server, and the client only needs to do minimal work to render the output of the presentation tier. Therefore, the pipeline was designed without adaptivity. Our new pipeline being necessarily adaptive must discretize the processing steps into operators forming a processing pipeline.

The operators identified are summarized in Table 5.1 in an example processing plan order. This processing plan is used for our evaluation in Chapter 6.

Operation Implementation	Description	Parameters
TrajectorySmootherOp	Detect errors and smooth out noise or jitter inherent in sensor measurements	**trajectory** - the trajectory to process **return** filtered trajectory
TrajectoryRepairerOp	Correct trajectories based on output of smoothing steps above	**trajectory** - the trajectory to process **filtered trajectory** - a filtered trajectory **return** repaired trajectory
ActivityRecognizerOp	Tag trajectories with components	**trajectory** - the trajectory to process **return** time sequence of activities matched

[Continued on next page]

Operation Implementation	Description	Parameters
SegmentReducerOp	From the output of activity recognition, reduce the amount of segments tagged erroneously (i.e., too short segments)	**activity sequence** - time sequence of activities matched **return** fixed activity sequence
NearbyFeaturesOp	Obtains nearby features from some spatial database based on trajectory	**trajectory** - the trajectory to process **return** list of features to consider
RegionizerOp	Align trajectories to features	**trajectory** - the trajectory to process **features** - nearby features to consider **return** time sequence of features matched
PersistenceOp	Save trajectory to persistent database for future queries	**trajectory** - the trajectory to process **activities** - matched activity sequence **features** - matched feature sequence

Table 5.1: Pipeline operators are listed in processing plan order. The operation implementation describes the implemented Java class name. The parameters describe the input operands and output result.

```
#Thu Aug 30 18:32:27 PDT 2012
system.network_bandwidth=30000
pipeline.order=0
pipeline.name=client
pipeline.priority=1.0
pipeline.algorithm=adaptive
```

Figure 5.5: Setting up *netem* with a flow rate of 200 kilobits per second and around 200 milliseconds of latency

Operating context

We use Hyperic's System Information Gatherer (SIGAR) [8] to gather information in a cross-platform manner. The Java virtual machine isolates running programs from architectural details, so SIGAR provides a Java Native Interface (JNI) bridge to system calls revealing information about the host machine. Through SIGAR, we can obtain the machine's load average (on Unix-like operating systems), the host CPU model and frequency, and available memory.

The network capability is also measured at the application level by evaluating the time for data to transfer between the client and server. Transfers are clocked against a synchronized clock on both the client and server (enforced by Network Time Protocol daemons with millisecond resolution). The transfer speed derived from the transfer time is an estimate of network load and congestion and is constantly updated against a moving average filter.

The combination of static and dynamic information is provided to the operator to make runtime decisions on processing, and in so doing, adopt principles of adaptivity.

Pipeline configuration

Finally, we provide a means to configure the pipeline through Spring XML beans files and properties files. The dependency wiring as well as database and network routing details are configured in Spring, however, the pipeline details are configured via Java properties files.

Figure 5.5 shows a sample configuration file for our adaptive pipeline. We include a *priority* parameter that weighs its own processing time described in Section 3.2.1 as P_c. Thus, a client would scale its $T_c(O_i)$ that it calculates with this parameter. This would allow for a client to prioritize energy usage over performance, or vice versa.

Chapter 6

Evaluation

6.1 Experimental Design

The experimental setup aims to demonstrate the viability of our adaptive pipeline in a variety of scenarios that may occur. The most common scenario will be that of multiple clients accessing a shared server. The server can be load-balanced by using various schemes such as a load balancer or even round-robin DNS. However, we evaluate the system with a single shared server and scale clients and network appropriately.

The server we choose is designed to be typical of a "cloud" server available from providers such as Amazon Elastic Compute Cloud or Google App Engine. It is provisioned as a Xen DomU — a virtual machine running on the Xen Hypervisor. The DomU share of resources is noted in Figure 6.1

For evaluation, it is difficult to control and simulate multiple mobile devices reliably, so we simulate multiple clients on standard personal computers instead (Figure 6.2). Multiple clients are run on multicore machines with the number of clients on each machine not exceeding the number of available cores. This ensures that there is little context switching and that

Processing	2 virtual CPUs — Intel i7-920 2.66 GHz
Disk	20 GB
RAM	1024 MB
OS	Debian 6.1
Network	Gigabit Ethernet

Figure 6.1: These cloud server system specifications we use for testing are representative of the products offered by common cloud computing vendors.

Processing	6 Core AMD Phenom II X6 1055T 2.8 GHz
Disk	1 TB
RAM	4096 MB
OS	Ubuntu 12.04
Network	Gigabit Ethernet

Figure 6.2: In our testing environment, up to 6 simultaneous clients run on machines using 6-core AMD processors. Each client's code is single threaded and uses a fraction of the available memory.

clients do not dramatically affect each other in processing.

6.1.1 Dataset

Our dataset is a randomly selected subset of 100 trajectories from a larger dataset recording snow sports. The trajectories represent activities such as skiing, snowboarding or cross country skiing and were recorded at many different snow resorts around the world. To the best of our knowledge, there has been no similar research done using a similar environment — most trajectories are captured in urban settings such as in cities and suburbs.

These trajectories are stored in GPX files which are parsed by each client process and loaded into memory. A GPX file holds the data described in 4.1. Trajectories are evenly distributed among all configured clients. The pipeline is setup with the operators as described in Section 5.2.3 and ordered into a processing plan created by the client.

6.1.2 Test scenarios

Adaptive pipelined work processing seeks to adaptively shift the burden of work as described in Chapter 3. The alternative is not to process it adaptively at all — that is, to process it entirely on the server or client. We test four different scenarios to evaluate the effectiveness of adaptive processing.

- Run as much as possible on the client. Thus, we only push work when the client is *unable* to perform the operation — it is an operation that can only be performed on the server. Thus, most of the load will be client-side, but it may incur heavy data transfer penalties.

- Push work as soon as possible to the server. In this scheme, no work, except for initial parsing of the trajectory is done by the client. The server will be doing all of the processing.
- Randomly choose a pipeline stage to push work to the server. In this scheme, operations are processed at random by client and server, and both do a share of the work.
- Employ the adaptive pipeline algorithm. Work will be allocated depending on the operating context and processing heuristics which allows the client and server to allocate work amongst themselves based on processing load and ability.

Each trajectory is individually processed through the pipeline. Each client processes trajectories iteratively and sequentially. However, the server processes requests simultaneously without limit (in practice, bounded by the worker thread limit, which is configured greater than the possible number of trajectories available). Each trajectory is individually timed from start to finish. Therefore, each trajectory time is an aggregate of processing and transfer times of each operator.

6.1.3 Test metrics

Pipeline performance

We will evaluate our processing pipeline in several ways. For a given test run, assume there are n jobs submitted and x_i is the time taken for the ith trajectory. Then, the total time for a given pipeline is the sum of the times of processing all trajectories through the pipeline.

$$totalTime = \sum_{i=1}^{n} x_i$$

We also look at the distribution of trajectory processing times by computing basic statistics such as the mean, median, standard deviation, minimum and maximum. For example, the average time is:

$$averageTime = \frac{totalTime}{n}$$

Fairness

However, to quantify the *fairness* — that is, whether the algorithm is assigning work such that each trajectory has the same share of system resources, we use a modified version of the Jain's fairness index [11] for bandwidth that returns a value from 0 to 1 where a completely fair system would be $J = 1$:

$$J(x_1, x_2, ..., x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} x_i^2}$$

6.1.4 Simulating network conditions

Using computer Ethernet as the network medium is not an accurate substitute for communications over mobile networks. In addition to evaluating performance over GigE, we also simulate mobile networks through the use of *netem* — a network emulation layer built into the Linux kernel [7]. Netem throttles the connection to EDGE with around 200 kilobits per second of bandwidth and around 200 ms of latency or around 400 ms of round trip latency (Figure 6.3).

6.2 Results

Our results show that our processing model is adaptive to changing client, server, and network loads. We will scale the size of our system by introducing more workers as well as look at the relative distributions of different pipeline schemes.

In Section 6.2.1, we simulate the scenarios identified in Section 6.1.2 on a client and server configuration with 6 clients. We follow this with Section

```
tc qdisc add dev eth0 root handle 1: htb default 1
tc class add dev eth0 parent 1: classid 1:1 htb rate 1000Mbps
tc class add dev eth0 parent 1:1 classid 1:3 htb rate 200kbps ceil \
        200kbps
tc qdisc add dev eth0 parent 1:3 handle 10: \
        netem delay 200ms 10ms distribution normal
tc filter add dev eth0 protocol ip parent 1:0 prio3 u32 match \
        ... flowid 1:3
```

Figure 6.3: Setting up *netem* with a flow rate of 200 kilobits per second and around 200 milliseconds of latency

	mean	median	std deviation	min	max
client	3973.6842	4098.0000	681.6079	1348.0000	6119.0000
server	4800.3825	4477.0000	2486.6226	1053.0000	10179.0000
random	3160.5439	3299.0000	988.9487	1043.0000	5125.0000
adaptive	2940.3158	3123.0000	1246.5026	1028.0000	6148.0000

Table 6.1: Basic statistics for different pipeline schemes using GigE with 6 clients, measured in milliseconds.

6.2.2 by evaluating this same scenario, but under EDGE-like mobile network conditions.

Next, we show our system's adaptiveness. Section 6.2.3 puts our adaptive processor against changing server loads by increasing the number of simultaneous clients. We also similarly scale the network capabilities in Section 6.2.4 to show that we obtain good results in changing network conditions.

6.2.1 Constrained processing only

In every situation, we expect that adaptive should perform better than all other schemes because it should optimize the allocation of work to minimize total processing time. The results confirm this. In the case of client-biased processing, the client is slower than the adaptive scheme because the client, while dedicated, is slower at certain operations. This is because the client's

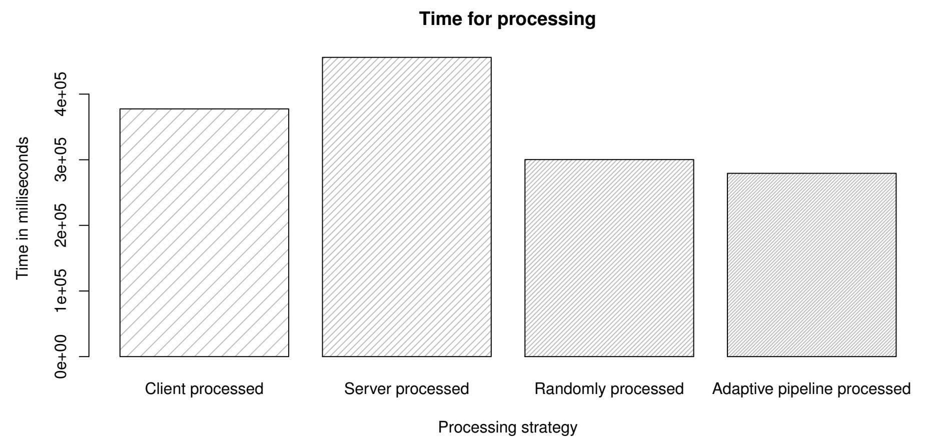


Figure 6.4: Processing comparison with 6 clients and 100 processed trajectories

processor is weaker than the server, or because the client requires remote fetching for data required for operation such as GIS feature matching. If the operation was performed on the server instead, the data fetch operation would be local instead. However, in the case of server side processing, the bottleneck becomes the server processing capabilities as it struggles to handle 6 simultaneous clients at the same time.

When reviewing the results, it is important to note that we did not tweak pipeline performance manually in any way. The adaptive algorithm automatically tuned performance to suit the processing capabilities and network capabilities of the client and server systems, as well as the nature of each operation such as the running time and selectivity.

6.2.2 Simulated mobile network

Under simulated EDGE network conditions, with 200 kilobits per second of bandwidth and 200 milliseconds of latency each way, the gap becomes smaller due to the constant effect of network affecting all schemes. We still expect the adaptive scheme to do better than all other schemes, but the network delays should mitigate factors in the gigabit network scenario such as server

priority	mean	median	std deviation	min	max
1.0	5229.724	5102.000	2156.927	1373.000	17954.000
0.9	5083.305	4642.000	2873.168	1426.000	32895.000
0.5	4729.768	4736.000	1573.417	1396.000	10952.000
0.3	4871.577	5024.000	1056.363	1610.000	9066.000

Table 6.2: Basic statistics of different priority parameters using simulated mobile network speeds with 6 clients, measured in milliseconds.

	mean	median	std deviation	min	max
client	5082.3642	5280.0000	880.4516	1917.0000	6980.0000
server	5676.0084	5695.0000	1919.3961	1419.0000	11959.0000
random	4760.0905	4747.0000	1417.5763	1402.0000	9739.0000
adaptive	4729.7684	4736.0000	1573.4173	1396.0000	10952.0000

Table 6.3: Basic statistics for different pipeline schemes under mobile network conditions with 6 clients, measured in milliseconds.

overloading because the rate of incoming requests is reduced due to network delays. Initially, when we tested this scheme, we discovered that the little network bandwidth made the algorithm delegate work at the earliest opportunity to try to minimize the amount of network data transferred. This consequently degraded into a pipeline that is almost entirely server processed which meant that the bottleneck once again became the server pipeline processor occasionally, which is represented by a high standard deviation and maximum. To solve this, we tweaked the processor priority parameter to discount client operators so that the algorithm should pick more operations to be executed on the client, even in spite of the increased network cost.

Table 6.2 shows that discounting client operations to 0.5, or half of their projected execution time, provides the optimal execution time for pipeline operations. Discounting further shows a decrease in performance, which can be attributed to the increased data costs involved. This configuration is compared to with the rest of our schemes in Figure 6.5.

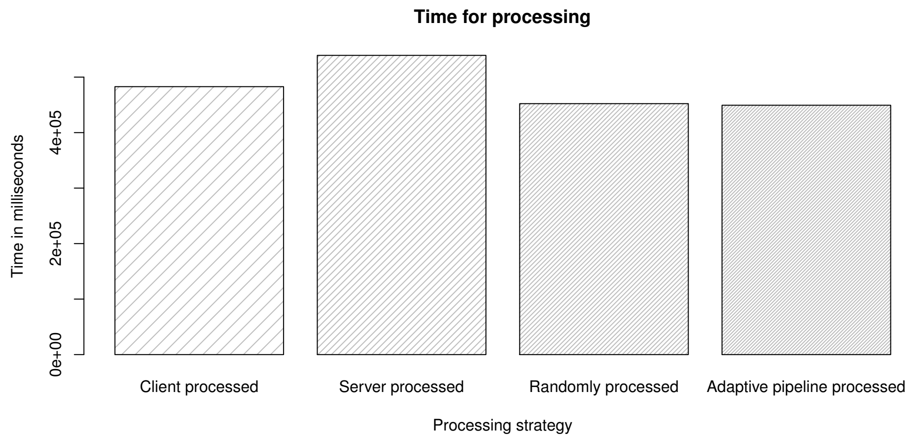


Figure 6.5: Processing results under mobile network conditions

Though we achieve better performance with our adaptive pipeline as demonstrated in Tables 6.1 and 6.3, statistical analysis of other measurements reveals some pitfalls. Compared to client only processing, since we have to use a shared server, the variance of performance increases such that in the worst case, we may take almost double the time with our adaptive pipeline compared to client only processing.

It is interesting that as we introduce more network delays and less bandwidth, the differences between the schemes lessen. Each node is less constrained by processing, so the algorithm must optimize network transfer instead. Of note is that random produces good average times compared to adaptive when network delays become extreme. This is a good scheme in many work processing algorithms (i.e., when pieces of work are roughly similar, allocate work randomly to worker queues) and does distribute the load well. However, our adaptive scheme may do better due to its more robust model of processing and network transfer.

6.2.3 Scaling server load and increasing number of clients

So far, we have only tried our adaptive scheme with a moderate work load (6 clients) where our adaptive model makes the most variable decisions on where to allocate work. However, in a light work load, we expect that an adaptive model should entirely process the work on the server. As server load increases, more work should be done on the client side. At the extreme end with many clients, the pipeline should be entirely client driven to mitigate server load as much as possible.

Figure 6.6 shows the scaling of work from 1 client to 32 simultaneous clients measured in average processing time for a trajectory. Processing work on the client has mostly level processing time as the majority of work is done on the client for which resources are not shared. However, it still rises by about 18 clients because the persistence operation which saves trajectories to the database is always performed server side regardless of processing scheme. At this point, the database server can no longer keep up with the incoming requests. The line representing server processing time increases immediately rises from 3 simultaneous clients up to 14 seconds per processed trajectory at 32 simultaneous clients. At low numbers of clients, the adaptive scheme tracks this line fairly well as server processing is still optimal to client side processing, but adaptively shifts part of the work to clients around 4-6 workers. It is also at this point that random does well as the adaptive processing algorithm is similar to randomly allocating pipeline operations and distributing the workload.

At high numbers of clients greater than 18 simultaneous clients, the adaptive pipeline more closely tracks the client scheme. The client scheme does better overall though, because of overhead in adaptive processing. Also, even when the adaptive model does all the work on the client, it will opportunistically try to shift work to the server from time to time. Therefore, the difference between the client-only processing line and the adaptive-only line can be thought of as the adaptive pipeline overhead.

We have shown that our adaptive pipelined work processing model is near-optimal for many different workloads. This is important in any distributed system, because workloads can change dramatically due to popularity surges and flash crowds. It is clear that no naive strategy can prevail for all kinds of workloads or system configurations — only an adaptive approach can optimally choose the best strategy.

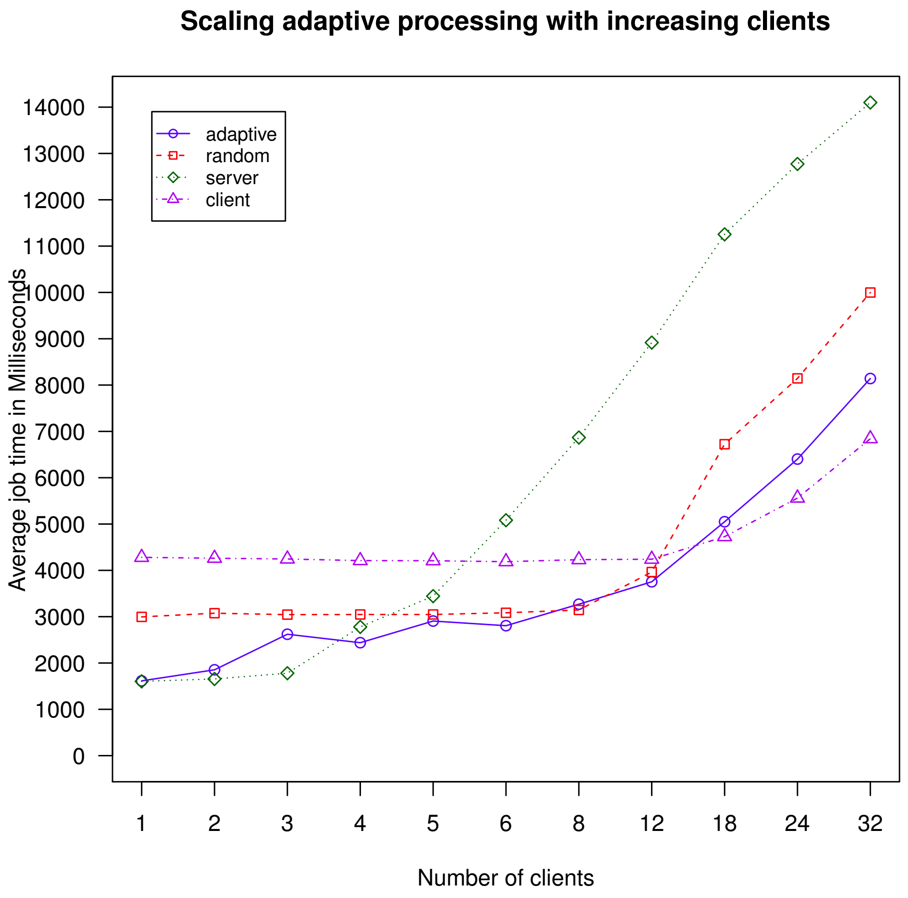


Figure 6.6: This graph shows the average processing time of a trajectory given an increasing number of simultaneous clients. Four different schemes are represented here: client-biased processing, server-biased processing, random allocation of processing, and our *adaptive pipelined work processing* model.

	fairness
client	0.9715157
server	0.7890245
random	0.9111071
adaptive	0.8481115

Table 6.4: Jain fairness for different pipeline schemes

6.2.4 Changing network conditions

Here, we simulate different network rate and delays with *netem*. We would expect that our model of network capabilities established in Section 3.2.3 should ensure our adaptive pipeline handles adverse network conditions well. At low network speeds, the system's network model is important to the adaptive processor since it should optimize the amount of data transferred for processing to be optimal. However, past a certain network rate, we would expect delays from processing to surpass delays from networking in the overall trajectory processing time. At this point, average times should remain unchanged for increasingly greater network rates.

Figure 6.7 depicts the performance of the different schemes for varying network loads. All tests were run with a network delay of 200 milliseconds. Of interest is the performance when the network has degraded to low speeds of 10—20 kbps. In such conditions however, the combined effect of a high latency link and low bandwidth causes high variability in transport speeds. This is due to effects of TCP congestion. Therefore, we see smaller differences between the schemes as network delays come primarily from congestion and resultant queueing delays, not transmission delays. We get a clearer picture at speeds of 50 kbps and above. The adaptive algorithm is consistently optimal or near optimal while the other schemes perform as expected from earlier experiments.

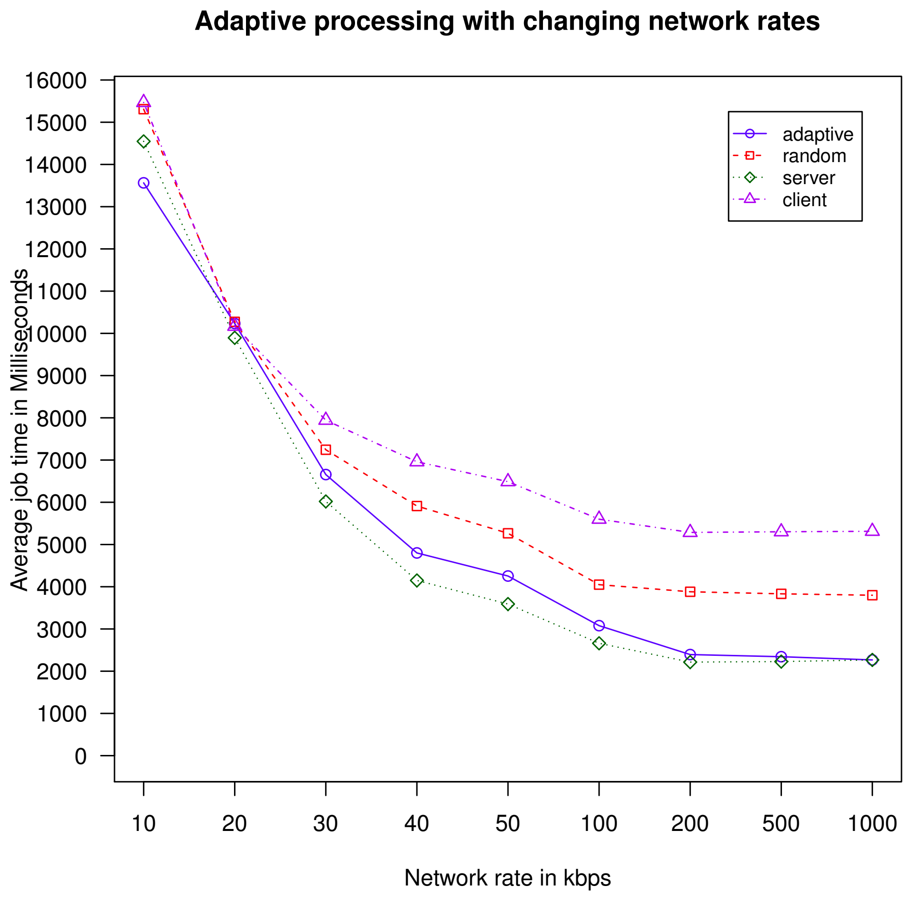


Figure 6.7: This graph shows the average processing time of a trajectory given an increasing network rate and 200 milliseconds of latency with a single worker. Four different schemes are represented here: client-biased processing, server-biased processing, random allocation of processing, and our *adaptive pipelined work processing* model.

	fairness
client	0.9709230
server	0.8975766
random	0.9186948
adaptive	0.9005509

Table 6.5: Jain fairness for different pipeline schemes under mobile network conditions

6.2.5 Analysis of fairness

In our analysis of fairness, we anticipate that a pipeline where the operations are entirely performed on the client would be fairest because the client has unique control over its own processor and is not shared. This explains the high > 0.97 index that is achieved by the client in both situations.

Table 6.4 lists the Jain Fairness Index for different scenarios (Section 6.1.3). While the adaptive pipeline clearly beats out the server pipeline in the robust network scenario, the gap between different schemes other than client is minimized to a single percentage point in the mobile network scenario. We have clearly opted throughout this thesis to optimize performance, and have only discussed fairness in passing. A future direction might be to extend this thesis along the lines of SEDA [27] and adjust the model for a trade-off between performance and fairness through the use of queues for each pipeline stage.

Chapter 7

Conclusion

The relevance of trajectory processing today is the direct result of the success of the GPS system introduced in 1973. Competing systems such as Galileo by the European Space Agency are being developed today, demonstrating the technological importance that satellite navigation has to modern society.

In this thesis, we have underscored the importance of trajectory processing and analyzed the systems and devices that are commonly used. In order to mitigate the pitfalls that are prevalent, we employed adaptive processing to design a system that performs optimally in a variety of real-world scenarios where traditional computing models break down. We demonstrated a number of different operations that a trajectory processing system would be expected to process. Our *adaptive pipelined work processing* is also resilient to failures and delays, and is simple for the application developer to use to implement a variety of different systems, and not limited to trajectory processing.

It is clear though, that there are other ways to optimize this model beyond performance. One such way is by measuring fairness. Though our model did not perform optimally in this regard, implementing queues between pipeline stages is a promising technique such as demonstrated by SEDA and would be a good direction for future work [27]. Another such optimization would be to use more powerful models to represent client and server performance. We used histograms and moving averages in order to approximate such behaviour, but there is no reason why more advanced techniques such as Kalman filtering and HMMs would not serve just as well. Furthermore, we have used a linear pipeline as our model of work processing here, but operations need not necessarily be sequentially executed. With the increasing availability of multi-core processors even on mobile devices, a more complex adaptive processing system could use concurrent workflows to accomplish quicker processing.

We hope that this thesis will motivate further research into trajectory processing and the closely related problem of processing with mobility. We support the development of systems which can extract intelligence from our data on the fly, toward future goals such as augmented reality. The numbers of GPS devices grow in the millions every year, and the decisions we make are ever increasingly data-driven.

Bibliography

[1] Peter Abeles. EJML: efficient-java-matrix-library. `http://code.google.com/p/efficient-java-matrix-library/`, 2012. [Online; accessed 31-August-2012].

[2] G. Bishop and G. Welch. An Introduction to the Kalman Filter. *Proc of SIGGRAPH, Course*, 8:27599–3175, 2001.

[3] S. Chandrasekaran, S. Ch, S. Madden, and M. Ionescu. Ninja paths: An architecture for composing services over wide area networks, 2000.

[4] P. Cudre-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 109–120. IEEE, 2010.

[5] E. De Lara, D.S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems-Volume 3*, pages 14–14. USENIX Association, 2001.

[6] Apache Software Foundation. Commons Math: The Apache Commons Mathematics Library. `http://commons.apache.org/math/`, 2012. [Online; accessed 31-August-2012].

[7] The Linux Foundation. `netem`. `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`, November 2009. [Online; accessed 31-August-2012].

[8] Hyperic. SIGAR API (System Information Gatherer). `http://www.`

hyperic.com/products/sigar/, 2012. [Online; accessed 31-August-2012].

[9] Recon Instruments. Heads-up Display Technology. http://www.reconinstruments.com/products/snow-heads-up-display, 2012. [Online; accessed 31-August-2012].

[10] A. Jain, E.Y. Chang, and Y.F. Wang. Adaptive stream resource management using kalman filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 11–22. ACM, 2004.

[11] R. Jain, D.M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *DEC research report TR-301*, 1984.

[12] M.T. Johnson. Capacity and complexity of HMM duration modeling techniques. *Signal Processing Letters, IEEE*, 12(5):407–410, 2005.

[13] R.E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 1960.

[14] J.A. Martín H, J. de Lope, and D. Maravall. Adaptation, anticipation and rationality in natural and artificial systems: computational paradigms mimicking nature. *Natural Computing*, 8(4):757–775, 2009.

[15] H.M.O. Mokhtar and J. Su. Universal trajectory queries for moving object databases. In *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*, pages 133–144. IEEE, 2004.

[16] D. Mosberger. *Scout: A Path-Based Operating System*. PhD thesis, University of Arizona, 1997.

[17] L.B. Mummert, M.R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. *ACM SIGOPS Operating Systems Review*, 29(5):143–155, 1995.

[18] K.P. Murphy. Hidden semi-markov models (HSMMs). *Informal Notes*, 2002.

[19] B.W. Parkinson and J.J. Spilker Jr. *Global Positioning System: theory and applications*, volume 1. American Institute of Aeronautics and Astronautics, Inc., 1996.

[20] K. Pentikousis. In search of energy-efficient mobile networking. *Communications Magazine, IEEE*, 48(1):95–103, 2010.

[21] L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[22] Alex Russell, Greg Wilkins, David Davis, and Mark Nesbitt. The Bayeux Protocol. `http://svn.cometd.com/trunk/bayeux/bayeux.html`, 2007. [Online; accessed 31-August-2012].

[23] M.S. Ryan and G.R. Nudd. The Viterbi Algorithm. *Warwick Research Report RR238*, 1993.

[24] F. Schmid, K.F. Richter, and P. Laube. Semantic trajectory compression. *Advances in Spatial and Temporal Databases*, pages 411–416, 2009.

[25] E. Seidel. Technology of high speed packet access (HSPA). *NOMOR Research White Paper*, 2006.

[26] SpringSource. Spring Framework. `http://www.springsource.org/spring-framework`, 2012. [Online; accessed 31-August-2012].

[27] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, October 2001.

[28] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.Y. Ma. Understanding mobility based on GPS data. In *Proceedings of the 10th International Conference on Ubiquitous Computing*, pages 312–321. ACM, 2008.

[29] Y. Zheng, X. Xie, and W.Y. Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Engineering Bulletin*, 33(2):32–40, 2010.

[30] Y. Zheng, L. Zhang, X. Xie, and W.Y. Ma. Mining interesting locations and travel sequences from GPS trajectories. In *Proceedings of the 18th International Conference on World Wide Web*, pages 791–800. ACM, 2009.

[31] Y. Zheng and X. Zhou. *Computing with spatial trajectories*. Springer-Verlag New York Inc, 2011.